Vývoj aplikací v jazyce Ruby

Bakalářská práce

Tomáš Kohout

Vývoj aplikací v jazyce Ruby: Bakalářská práce

Tomáš Kohout

Vydáno duben 2008

Anotace

Bakalářská práce se zabývá skriptovacím jazykem Ruby a jeho nadstavbou pro vývoj webových aplikací Ruby on Rails. Autor se věnuje charakteristickým rysům jazyka Ruby na různých příkladech. Objasňuje vazbu Ruby na framework Rails a popisuje techniky a postupy vývoje aplikací v tomto programovacím jazyce. Dále se zabývá vývojovými nástroji a prostředími vhodnými pro tvorbu v Ruby (Ruby on Rails). Pokouší se o srovnání s ostatními programovacími jazyky a shrnuje možnosti programátora v Ruby. Součástí práce je návrh a vývoj konkrétní aplikace v jazyce Ruby – webový server sudoku s kompletní dokumentací, vývojovými návrhy (UML schémata) a kompletními zdrojovými kódy.

Abstract

This work is engaged in a script language Ruby and its enlargement Ruby on Rails for developing web aplications. It is attended to features of Ruby language in different examples. It illustrates bindings between Ruby and Ruby on Rails framework. It describes techniques and procedures for developing aplications in this language. Furthermore it is focused on development tools and suitable environments for a production in Ruby (Ruby on Rails). It tries to compare Ruby with the other programming languages and it sums up the facilities of programmers in Ruby. Part of this work is a project of concrete aplication in Ruby language – web server of sudoku with complete documentation, developing schemes (UML) and complete source codes.

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce doktoru Ichovi za jeho trpělivost a přístup k mé práci. Jeho zkušenosti a nadhled mi hodně pomohly při tvorbě.

Dále bych chtěl poděkovat své rodině, za pomoc při vývoji webového serveru sudoku, kdy odhodlaně testovali a soupeřili s mými algoritmy matematického modelu sudoku.

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě, fakultou elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne 24. dubna 2008

Tomáš Kohout

Obsah

Seznam obrázků

Kapitola 1. Úvod

1.1. Poznámka k formátu bakalářské práce

Bakalářská práce byla vytvořena v šabloně **docbook 4.3**. Byla psána jako čisté **XML** s pomocí tagů docbook. Je použito přímo šablony BOOK pro tvorbu knih (i se všemi typografickými pravidly, která platí pro český text). Samotný text jsem psal v XML editoru **XML Mind**. Soubor XML bakalářské práce je také dostupný na CD v příloze. Výstupní publikační formát jsem zvolil PDF. Pro transformaci práce z XML do PDF jsem použil nástroj **XEP** (volně dostupný preprocesor textu). Renderovací nástroj XEP jsem počeštil utilitou od pana Koska dostupnou na internetu (http://www.kosek.cz/xml/db/inst.html), tím jsem docílil českých popisů (místo Chapter kapitola, místo Content obsah, atd.) i českého dělení textu.

1.2. Cíle bakalářské práce a její přínos

Na úvod bych především přiblížil jaké cíle jsem si stanovil v této bakalářské práci. Jejím hlavním cílem a přínosem by mělo být přiblížení jazyka Ruby a frameworku Ruby on Rails programátorské komunitě, která zatím nemá o RoR dost velké povědomí. V této práci se zabývám jak zásadními principy, které potřebujete pro programování v Ruby on Rails, tak i zajímavými detaily, na které jsem narazil při realizaci vlastní aplikace. Tyto detaily se mohou stát užitečnými pro další programátory, kteří se rozhodnou jít v mých stopách a začít vyvíjet dynamický web v prostředí Ruby on Rails. V dnešní době je k sehnání dost informací o jazyce Ruby i o Ruby on Rails, ale bohužel se převážně jedná o útržkovité informace, které bývají často vytrženy z kontextu a programátor musí často hledat na mnoha místech, než se dobere kvalitní odpovědi. Existuje sice dokumentace jazyka Ruby, ale srovnávat jí s dokumentací např. pro Javu by bylo holé šílenství. Podpora a dokumentace Javy je kvalitativně i kvantitativně na mnohem vyšší úrovni. Bakalářská práce má ukázat i vývoj konkrétní aplikace od prvního návrhu po kompletní zdrojový kód, který najdete v příloze na CD. Tento kód by měl názorně začínajícímu programátorovi předvést konstrukce jazyka Ruby. Nejlépe se učí na konkrétních příkladech a ukázková aplikace byla zvolena tak, aby pokryla širokou škálu možností požadavků na dynamický web. Na ukázkách kódu se budu snažit rozebrat jednotlivé detaily a na příkladech jednoduchých aplikací nebo návrhů ukázat sílu Ruby on Rails. V této práci by měl programátor najít odpověď na to, proč vlastně používat Ruby on Rails, co je na něm tak zajímavé, zvláštní nebo efektivní a proč patří k tak oblíbenému frameworku ve světě.

1.3. Místo Ruby on Rails v dnešní změti webových frameworků

Převzato z internetového článku na weblogu o Rails: citace zdroje [1]

Framework pro vývoj webových aplikací a stránek Ruby on Rails si za poslední dva tři roky získal takovou slávu, že jedni jsou nadšeni zcela nad míru běžně obvyklou a kážou Rails kudy

chodí, druzí mluví o „pouhé módě“, „skvělém marketingu 37 Signals“ a „zbytečném hype“. Z pohledu na významné vývojářské blogy či weby nebo videa, kde v patnácti minutách naprogramujete kompletní blog člověk rozhodně získá dojem, že Rails jsou mýtická „stříbrná kulka“ softwarového vývoje, technologie, jejíž pouhé použití či nasazení několikanásobně zvyšuje produktivitu. A k dovršení všeho na Ruby on Rails běží nejslavnější aplikace a stránky současného webu: Basecamp, Shopify, Cork'd, A List Apart či Twitter. Není třeba si tedy klást (řečnické) otázky jako „Je Ruby on Rails stříbrná kulka?“ a podobně, ale můžeme rovnou vyložit karty. Řekněme to narovinu:

Jestliže se zabýváte vývojem pro web, Ruby on Rails je to nejzajímavější, co se za poslední dva roky na tomto poli objevilo. Rails je killer app webového vývoje.

Panebože! Slyším ty výkřiky! Ne Ajax? Ne PHP 5? Ne Script.aculo.us? Ne … Zend framework? Ne Adobe Apollo... tedy AIR? Ne. A nejenom proto, že Rails v sobě Ajax i Script.aculo.us zahrnují (a v jistém smyslu, k němuž se vrátíme, teprve akcelerují, neboli „staví na koleje“). Rails jsou jedinečné (a v tomto smyslu „nejzajímavější“) proto, že kromě mnoha ryze technických konceptů a inovací vnesly do webového vývoje zásadní intelektuální koncepty a nové myšlenky. To je pohled, který v debatách ohledně Ruby on Rails pravidelně zapadá. Debaty ohledně Rails totiž trpí známým syndromem „to není nic nového pod sluncem“. Poučná je v tomto smyslu debata k článku „Proč bylo Ruby on Rails možné napsat pouze a jedině v Ruby?“. V ní se diskutující pravidelně vracejí k objevné myšlence, že „to samé“ je možné udělat v Pythonu nebo v Perlu, že Model-View-Controller není nic nového, že lepší než Rails je framework Django pro Python nebo Catalyst pro Perl. A další variace: to umí .NET/Java/atd. už dávno a líp, a navíc mají IDE, a striktní typování. PHP je rozšířenější, a rychlejší, a levnější na provoz. A s „free-hostingem“. Jenže na webu se nyní o ničem nemluví s takovým nadšením jako o Rails, a jasná volba pro všechny obdivované start-upy není ani .NET, ani Python. Ani PHP. Tato diskuse totiž posuzuje Rails z omezeně technologického pohledu. A z omezeně technologického pohledu je Ruby on Rails skutečně jen další MVC webový framework. Co však Rails činí zajímavé a jedinečné jsou netechnické (ve striktním slova smyslu), intelektuální a sociální aspekty, kterými se chci zabývat v těchto článcích. Právě ty ve vývojářích vyvolávají pocit „všechno v Rails a už nikdy jinak“ a právě ty vyvolaly mohutnou vlnu nadšení a nové energie pro vývoj na webu.

Kapitola 2. Rozdělení skriptovacích jazyků

V kapitole je volně čerpáno z následujích zdrojů (Použita pouze fakta o skriptovacích jazycích, volně přeloženo autorem): zdroj [2], [3], [4]

2.1. Interpret versus kompilace

Nejprve si musíme odpovědět na otázku, co je to skriptovací jazyk, čím se vyznačuje a čím se liší od ostatních programovacích jazyků. Skriptovací jazyk není kompilován ale interpretován. V praxi to znamená, že nepotřebujeme žádný kompilátor, který převede zdrojové kódy do podoby binární spustitelné aplikace. Potřebujeme ale INTERPRETER, což je speciální aplikace, která naprogramovaný skript vykonává přímo ať už na serveru nebo někde jinde. Zdrojový kód je vykonáván řádku po řádce. To ovšem znamená, že interpreter musí před před každým spuštěním znovu analyzovat zdrojový kód. Najít syntaktické chyby, atd. Proč tedy skriptovací jazyk? Bez potřeby kompilace můžeme kód rychle a snadno vyvíjet. A pokud známe přesný cíl programu a víme, že nepotřebujeme nasadit robustní kompilovanou aplikaci, vyplatí se použít skriptovací jazyk. Tyto jazyky se převážně prosadily v prostředí webových aplikací.

2.2. Shell skripty

Pod přímou definici skriptovacího jazyka spadají i takzvané SHELL skripty a různí démoni pracující automaticky na pozadí. Tyto vlastnosti najdeme v Unixovém shellu, tedy příkazové řádce nebo v MS DOSU jako COMMAND.COM. Tyto programy provádějí přímo příkazy psané z klávesnice, žádná kompilace, linkování. Přímý vstup z klávesnice je prováděn řádek po řádku za účasti dalších běžících činností na pozadí. Mezi tyto programy patří například: AppleScript, bash (který najdeme v Linuxu), csh, DCL (u OpenVMS), ksh, cmd.exe (u Windows NT, CE a OS/2), zmiňovaný COMMAND.COM (u MS DOSU a Windows 9x), WinBatch, atd.

2.3. GUI Skriptování - MAKRO jazyky

Společně s příchodem grafického uživatelského rozhraní alias aplikačních oken přišla i potřeba specializovaných jazyků, které budou obsluhovat přímo okna, menu a jednotlivá tlačítka. Tyto jazyky jsou používány k automatickému opakování akcí, konfiguraci a obnovování stavu komponent. Měly by být schopny řídit jakoukoliv GUI aplikaci, ale v praxi záleží ještě na konkrétním operačním systému. Jako příklad můžeme uvést funkci AutoHotkey, což je free, open source makro, které provádí jednotlivé úkoly při stisku klávesové zkratky (ve Windows) od vývojáře Chrise Malletta. Tento software je napsán pomocí skriptovacího jazyka Autolt2. Autolt je volně šiřitelný jazyk od Microsoftu, který je určen především k vytváření takovýchto a podobných automatických maker pro Microsft Windows. Pro Unix existuje nástroj Expect (od vývojáře Dona Libese), který je rozšířením skriptovacího jazyka Tcl (Tool Command Language). V Linuxu slouží pro interaktivní aplikace do X11 GUI jako jsou telnet, ftp, ssh, atd. Pro Apple a Mac OS X existuje aplikace Automator, která se specializuje na point-and-click

nebo drag-and-drop události. Tento program pak funguje napříč celou řadou aplikací v Mac OS X.

2.4. Specifické jazyky

Mnoho programů obsahuje osobité skripty ušité na míru potřebám uživatele aplikace. Například v oblasti počítačových her se tyto skripty používají pro generování a ovládání tzv. NPC - Non Player Characters, což jsou postavy nebo objekty, které nejsou ovládány uživatelem, jsou pouze součástí prostředí s nímž interagují. Například pro tyto NPC ve hře Quake je používán speciální skriptovací jazyk QuakeC. Mezi další takovéto speciální jazyky patří Action Code Script, ActionScript, HyperTalk, Lingo, LotusScript a další.

2.5. Jazyky pro web

Důležitým typem skriptovacích jazyků jsou ty, které jsou určené pro tvorbu dynamických webových stránek. Převážná většina těchto moderních jazyků, která je určená především pro web si už ale poradí i se všemi ostatními problémy a dají se použít dokonce ke stejným účelům jako standardní jazyky C, Java. Pro obecné účely.

Rozlišujeme dvě kategorie těchto jazyků. Do první spadají jazyky, jejichž kód je interpretován přímo na straně serveru. Klient pošle žádost, na serveru se spustí žádaný skript a server vrátí do prohlížeče odezvu, kterou vyprodukoval uložený skript. K těmto jazykům se řadí především nejznámější PHP, SMX, ColdFusion, ale můžeme sem zařadit i jazyk XSLT, který se stará o transformaci XML.

Na straně klienta se pak jedná o jazyky, které pracují přímo s prohlížečem, reagují na vstupy přímo od uživatele a provádí na ně požadovanou reakci. Tyto akce se dějí nezávisle na serveru. Jedná se o různé kontroly vyplnění formuláře, validity dat, atd. Do této kategorie patří hlavně JavaScript nebo VBScript

2.6. Jazyky pro zpracování textu

Tyto jazyky patří mezi nejstarší používané skriptovací jazyky. Jsou zaměřené na automatické úkoly, které mění konfiguraci pomocí textových souborů nebo zapisují události do logovacích souborů. Najdeme je převážně na systémech typu Unix. Patří sem i známý jazyk Perl, který se ale ve svém vývoji pustil mnohem dál, o čemž se zmíním v historii skriptovacích jazyků. Dále sem patří jazyky AWK, sed a také XSLT, které přece jen pracuje s XML, tedy s čistým textovým souborem.

2.7. Jazyky pro obecné účely

Některé jazyky jako například Perl začínaly jako skriptovací jazyky, ale později se vyvinuly v dokonalejší programovací jazyky, které se dají použít pro jakékoliv obecné účely. Tyto jazyky jsou převážně interpretované, ale vykazují znaky vyšších programovacích jazyků jako je práce s objekty, automatická správa paměti, některé z nich jsou dokonce dynamické. Tato skupina

trochu uniká z pojmu skriptovacích jazyků i když sem samozřejmě patří. Jazyky jako je Perl, Python, SmallTalk, Ruby, Groovy, Tcl už nejsou primárně určené pro vývoj webových aplikací a pro zpracování jednoduchých skriptovacích úloh, ale jsou požitelné i pro rozsáhlé a robustní aplikace.

2.8. Rozšiřující jazyky

Řada jazyků je vytvořena k nahrazení specifické aplikace nebo její části. Využívají vložitelnosti do aplikačních programů. Aplikační programátor vloží (například do kódu v jazyce C) tzv. "HOOKS" - háky, které označují místo, kde přebírá řízení aplikace skriptovací jazyk. Tyto jazyky slouží ke stejným účelům jako SPECIFICKÉ JAZYKY ale s výhodou přenášení různých dovedností, schopností od aplikace k aplikaci. Do této kategorie patří například Ch (interpret jayka C, C++), ECMAScript, EOS, Squirrel, Z-Script.

2.9. Ostatní jazyky

Do této skupiny můžeme zahrnout všechny ostatní jazyky, které se nepodařilo zaškatulkovat do ani jedné předchozí kategorie. Jsou to jazyky určené přímo ke konkrétním úkolům a přímo pro tyto úkoly vytvořené. Například KonsolScript, který je open source a je určen k vývoji 2D desktopových her. Nebo CobolScript, založený na syntaxi jazyka COBOL, určený ke konverzi dat a řešení skriptování na straně serveru.

Kapitola 3. Historie a charakteristiky skriptovacích jazyků

V kapitole je volně čerpáno z následujích zdrojů (Použita pouze fakta o skriptovacích jazycích, volně přeloženo autorem): zdroj [2], [3], [4]

3.1. Perl

Mezi nejznámější a nejstarší skriptovací jazyky(je kompilovaný, ale vykazuje i znaky skriptovacího jazyka) mezi profesionály patří rozhodně jazyk Perl. Je to dynamický programovací jazyk. Jeho historie sahá do roku 1987, vytvořil ho programátor Larry Wall, který tehdy pracoval pro firmu Unisys. Uvedení jazyka Perl znamenalo jeho masové rozšíření během několika málo let. Jméno Perl vzniklo z definice jazyka: for Practical Extraction and Report Language. V roce 1988 už vznikl Perl 2 a o rok později Perl 3, který se chlubil podporou binárních datových toků. Do roku 1991 se dokumentace Perlu omezovala pouze na jednu manuálovou stránku. V tomto roce ale vznikla publikace **Programming in Perl**, která je známá především pod jménem "Camel Book". Právě odtud - z přebalu této knihy pochází logo Perlu, které se táhne od začátku devadesátých let, tím logem je právě velbloud. Tato kniha byla koncipována v podstatě jako referenční příručka k jazyku Perl. V roce 1993 přichází na scénu Perl 4 , ale Larry Wall pracuje už na pětce, která je uvedená v říjnu téhož roku. Tato verze obsahuje značné změny v interpreteru. Podporuje objekty, reference a lokální proměnné, zvané MY VARIABLES. V současné době se Perl dočkal podpory pro UNICODE a samozřejmě pro práci s vlákny.

Do dnešních dnů se Perl převážně používá pro tvorbu webových CGI skriptů. CGI znamená Common Gateway Interface. Označuje se tak rozhraní, které se skládá z externího aplikačního softwaru s informačním serverem a web serverem. Ten prochází žádosti od uživatele z webového prohlížeče a posílá je externí aplikaci. Aplikace pak pošle zpět odpověď přes server do webového prohlížeče klienta. Historie CGI se datuje zhruba od roku 1993. Rob McCool z firmy NCSA se zabýval implementací v HTTPd - originálním řešením webového serveru od NCSA. Vývoj NCSA HTTPd se zastavil v roce 1998, ale myšlenka a část zdrojových kódů přešlo do mnohem známějšího projektu Apache. CGI skripty používá celá řada webových serverů. Jako příklad může posloužit Wikipedia, volná internetová encyklopedie.

Perl přišel v době, kdy se ostatní jazyky jako Fortran a C snažily o efektivní využití drahého hardwaru, s myšlenkou na efektivní využití spíše ze strany programátorů. Nabízí mnoho možností, jak zjednodušit a zrychlit složité úkoly. Mezi ně patří automatická správa paměti - dnes známá jako Garbage Collector, který je obrovským pomocníkem řady programátorů. Dále dynamické typování. Dynamické typování znamená použití proměnných bez deklarace typu. Typ se proměnné přiřadí až při jejím prvním použití. Tato vlastnost hodně zjednodušila práci programátorů, ale její výhoda je trochu relativní, protože snižuje přehlednost kódu a také snižuje schopnost hledat chyby v kódu. Perl používá i jednoduchý typ string, který klasické C nezná. Dále pak listy, tedy seznamy, regulární výrazy, hash tabulky a speciální funkci eval(). Tato funkce umožňuje poslat obyčejný string jako by to byl výraz či příkaz a vrátit výsledek. Tato funkce bývá převážně výsadou skriptovacích jazyků narozdíl od jazyků kompilovaných.

Perl je vyvíjen jako kompletně volně šiřitelný software pod GNU General Public License a je dostupný pro převážnou většinu operačních systémů. Převládá na oprečních systémech typu Unix. Perl může být dokonce zkompilován C kompilátorem na všech Unix-like strojích. Pro Windows existuje binární distribuce ActivePerl.

3.2. Python

Patří mezi HIGH LEVEL programovací jazyky. Upřednostňuje úsilí programátora před úsilím počítače a jeho prioritou je čitelnost na úkor rychlosti a výmluvnosti (stručnější jazyk). Python byl koncipován na konci 80. let. Jeho tvůrce Guido van Rossum z Nizozemí tehdy pracoval pro mezinárodní výzkumný institut matematiky a výpočetní techniky v Amsterdamu. Programovací jazyk Python nebyl jeho prvním dílem, podílel se i na jazyku ABC, na Algolu 68 nebo na MonetuDB. V roce 1991 zveřejnil Rossum kód Pythonu. Uměl pracovat se třídami, dědičností a hlavně dokázal obsluhovat výjimky. Datové typy obsahovaly jak seznamy (LIST) tak slovníky (DICT). Také měl modulární systém, založený na myšlence jazyka Modula-3. Převzal hlavně výjimkový model, tzv. blok TRY ... CATCH, který používají i dnešní moderní programovací jazyky jako například Java. Verze Pythonu 1.0 vyšla v lednu 1994. Mezi hlavní znaky patřil **lambda kalkulus** - formální systém pro řešení a definici výpočetních funkcí (Lambda byla představena matematiky A. Churchem a S.C. Kleenem už ve 30. letech 20. století). Dále nástroje map, reduce a filter. Od verze 1.4 získal Python několik nových znaků. Vestavěnou podporu komplexních čísel, jednoduchou formu ochrany dat (data hiding, který se v moderních jazycích vyvinul do podoby ENCAPSULATION - zapouzdření). Rossum poté vyšel vstříc všem programátorům s iniciativou Computer Programming for Everybody (CP4E). Snažil se o rozšíření programování i mezi veřejností, tedy programátory amatéry. K tomuto účelu mu velice dobře posloužil jazyk Python, který neměl sloužitou syntaxi ani sémantiku a byl plně použitelný. Tento projekt byl dokonce zaštítěn agenturou DARPA (Defense Advanced Research Projects Agency), která je součástí Ministerstva obrany USA. V roce 2000 vývoj Pythonu pokračoval už v týmu pod BeOpen PythonLabs. Verze Pythonu 2.0 byla první a jediná vydaná pod BeOpen.com, Rossum se společně s dalšími vývojáři připojil k Digital Creations. Python 2.0 si své nejlepší vlastnosti propůjčoval od funkcionálního programovacího jazyka Haskell, podobná byla i syntaxe. Dočkal se i garbage collectoru, tedy automatické správy paměti.

Mezi největší projekty, pro které byl použit jazyk Python patří aplikační server Zope nebo Mnet, což je peer to peer server, úložiště sdílených dat. Nejznámější aplikací napsanou v Pythonu je BitTorrent.client. Python je často využíván jako skriptovací jazyk v 3D animačních balíčcích. Například v softwaru Maya nebo Blender. Python je standardní komponentou unixových operačních systémů, FreeBSD i NetBSD. Najdeme ho i v MAC OS X. Distribuce linuxu Gentoo ho dokonce používá pro práci s portage stromem a nástrojem EMERGE, který automaticky dokáže najít stáhnout, zkompilovat a nainstalovat aplikace pomocí balíčků.

Vývoj jazyka Python se ubíral směrem k verzi 3.0 (v době psaní bakalářské práce), která vyšla v srpnu roku 2007. Tato verze přerušila zpětnou kompatibilitu s řadou 2.x. Podle Guida van Rossuma nejsou požadavky na zpětnou kompatibilitu. Mezi dalšími novinkami má být i podpora UNICODE. Tato verze 3.0 je vývojáři nazývána také Python 3000 (Py3K).

3.3. PHP - Personal Home Page Tools

Patří mezi poměrně mladé jazyky. PHP napsal Dánský programátor Rasmus Lerdorf v roce 1994 jako sadu binárních kódů v jazyce C. Chtěl tím nahradit sadu Perl skriptů, které používal k udržování svých soukromých webových stránek. V roce 1995 vypustil Lerdorf do světa Personal Home Page Tools. Po kombinaci s jeho vlastním interpreterem vzniklo PHP/FI, známé jako PHP verze 2. Dva izraelští programátoři Zeev Suraski a Andi Gutmans přepsali v roce 1997 parser a vzniklo PHP 3. Změnil se také význam názvu PHP - Hypertextový preprocesor. Oficiálně bylo PHP 3 vypuštěno v červnu 1998. Suraski a Gutmans začali přepisovat jádro PHP jako Zend Engine, na což si založili vlastní firmu Zend Technologies. V květnu 2000 připravili PHP 4 na Zend Enginu 1.0. Červenec 2004 znamenal příchod PHP 5 na Zend Enginu II, které obsahovalo několik nových vylepšení.

Mezi tato vylepšení patří především zlepšení podpory objektově orientovaného programování, zvýšení výkonu, mnohem lepší podpora MySQL databáze, přidaná podpora pro SQLite, integrovaná podpora pro SOAP technologii, datové iterátory, a práce s výjimkami.

Hlavní důraz jazyka PHP je především kladen na vývoj serverových aplikací, tedy skriptů, běžících na straně serveru. V těchto aplikacích je plně srovnatelný s Microsoft ASP .NET nebo JavaServer Pages, nebo třeba právě i s Ruby on Rails. Ve webovém průmyslu se především začal používat model LAMP - Linux, Apache, MySQL, jako Programovací jazyk se zde právě nejvíce prosadilo PHP. Toto masové rozšíření znamená, že přes 19 milionů internetových serverů má nainstalováno PHP.

Na straně klienta nabízí PHP spojení s GUI knihovnami jako je třeba GTK+ (PHP-GTK) a s textovými knihovnami (např. NCURSES). Tyto knihovny umožňují vývoj aplikací na straně klienta s grafickým uživatelským rozhraním a to napříč různými platformami.

3.4. ECMAScript

Skriptovací jazyk standardizovaný firmou Ecma International podle specifikace ECMA-262. Jazyk je vhodný pro široké použití na webu a je také uváděn jako JavaScript nebo JScript podle hlavních implementací této specifikace.

JavaScript byl vyvinut Brendanem Eichem pod jménem Mocha, později jako LiveScript a konečně JavaScript. V roce 1995 uvedl Sun Microsystems a Netscape příchod JavaScriptu v tisku, o rok později vyšel Netscape Navigator 2.0, podporující JavaScript. Po obrovském úspěchu JavaScriptu jakožto jazyka pro klientské skripty přišel Microsoft s kompatibilním jazykem nazvaným JScript, který přinášel nové funkce a opravoval JavaScript pro problém Y2K (přechod roku 2000). JScript byl obsažen v Internet Exploreru 3.0. ECMAScript je tedy standardem podle ECMA-262, od kterého jsou odvozené (a rozšiřují ho) jak JavaScript tak JScript.

Pokud se podíváme na jednotlivé prohlížeče, je jasně vidět konkurenční boj mezi Netscapem a Microsoftem. Browsery založené na Gecku, tedy Mozzila Firefox používají JavaScript. Internet Explorer od Microsoftu razí svou cestu s JScriptem a Opera spojila JavaScript i JScript.

Nová verze podle ECMA-262 (čtvrtá série) je zpětně kompatibilní s ECMAScriptem 3 a zároveň přináší podporu objektových tříd, balíčků, namespaců a iterátorů.

3.5. ActionScript

Patří mezi další jazyk založený na ECMAScriptu, je primárně určen pro vývoj webů a softwaru s použitím Adobe Flash přehrávače od firmy Macromedia (v současnosti vlastněná firmou Adobe). ActionScript je vytvořen pro řízení jednoduchých 2D vektorových animací, vytvořených v Adobe Flash. Poslední verze umožňuje vytvářet jednoduché webové hry a také přenášet streamované video a audio.

Historie ActionScriptu se pevně váže k nástroji Macromedia Flash. Jednoduché příkazy umožňovaly přiřadit akci k tlačítku nebo ke snímku. Zpočátku to byly pouze funkce jako PLAY, STOP, GetURL, GOTO a PLAY. S verzí Flash 4 v roce 1999 tyto funkce dozrály do podoby malého skriptovacího jazyka. Tento jazyk obsahoval proměnné, výrazy, operátory, podmínky if a cykly. V současné době se vývoj zastavil na verzi ActionScriptu 3.0 a přehrávače ve verzi Flash Player 9.

Poslední verze ActionScriptu nabízí podporu namespaců, balíčků, regulárních výrazů. Samozřejmě také dědičnost, prototypy a výjimkový model.

3.6. JavaServer Pages

JSP je javovská technologie, která umožňuje vývojářům dynamicky generovat HTML, XML nebo jiné dokumenty v odezvě na žádost webového klienta. JSP je kód Javy plus předdefinované akce, tzv. XML-like tagy, které se také nazývají JSP akce, které se používají pro vyvolání vestavěné funkcionality. Knihovna tagů poskytuje platformovou nezávislost. JSP je kompilováno do podoby Java Servletu prostřednictvím JSP kompilátoru. Ten je pak překompilován samotným kompilátorem Javy. JSP kompilátor může také rovnou vytvořit byte kód pro servlet přímo.

JSP pochází stejně jako Java od Sun Microsystems, konkrétně na JSP a servletech pracoval Anselm Baird Smith a později Satish Dharmaraj. Nejnovější verze JSP jsou v současné době vypouštěny do světa jako součást balíku Java EE - Enterprise Edition.

3.7. Tcl - Tool Command Language

Přes zkratku Tcl se vžilo označení "tickle", které má hodně zajímavých významů. Tento skriptovací jazyk vytvořil profesor výpočetní techniky z kalifornské univerzity v Berkeley. K němu vytvořil i GUI Toolkit takzvaný Tk. V této kombinaci se Tcl/Tk stalo mocným nástrojem pro své jedinečné vlastnosti. Jazyk, který není těžký na zvládnutí a který může být velice efektivní, pokud se správně použije.

Všechno v tomto jazyce je chápáno jako příkaz. Používá se tzv. Polská notace (prefixová) - operátory se umisťují na levou stranu před operandy. S tímto systémem přišel kolem roku 1920 polský logik Jan Lukasiewicz a proto se někdy nazývá Polská notace, Mezi další výhody jazyka Tcl patří dynamičnost, vše může být dynamicky předefinováno nebo přepsáno. Všechny datové

typy mohou být chápány jako stringy a může tak s nimi být nakládáno. Tcl má velice jednoduchá syntaktická pravidla. Od roku 1999 je podporován Unicode. Tcl původně nebyl navržen jako objektově orientovaný jazyk, ale existují jakési nástavby, které umožňují na jednoduché bázi pracovat s objekty, např. STOOOP (Simple Tcl only object oriented programming).

Mezi příklady použití Tcl můžeme najít populárního IRC Bota Eggdrop. V roce 1993 přišel Eggdrop (autoři: Robey Pointer a Jamie Rishaw) jako náhrada za cEvin. Byl napsán v jazyce C a ve skriptovém rozhraní se prosadil právě Tcl.

3.8. Smalltalk

Mezi starší a velice dobře známé jazyky patří Smalltalk. Tento jazyk je kapitolou sám o sobě. Původně byl vytvořen hlavně pro výukové účely. Zasadil se o to tým programátorů z Xeroc PARC (Palo Alto Research Company, Kalifornie) Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace a další v průběhu 70. let 20. století. Velký vliv při vývoji si připsaly jazyky Lisp, Logo, Sketchpad a Simula. První verze byla vypuštěna pod názvem Smalltalk-80 a začal se široce prosazovat od 80. let. Vycházel z původních konceptů Smalltalk-71 až 76, ale Smalltalk-80 byla první verze vypuštěná vně společnosti PARC. V průběhu 90. let prosazovaly Smalltalk dvě firmy PARCPlace Systems, která se zaměřovala hlavně na OS Unix a DigiTalk, která dodávala Smalltalk pro Microsoft Windows a OS/2. Bojovaly i s nedostatky Smalltalku, kterými rozhodně byla slabá podpora pro SQL databáze, se slabším celkovým výkonem v runtimu i čitelností kódu v distribuovaných aplikacích. V roce 1995 se obě společnosti sloučily jako ObjectShare. Po roce 1999 byla společnost rozprodána a dnes je součástí Cincom Smalltalk.

Smalltalk-80 znamenal průlom v objektově orientovaném programování. Všechno se stalo objektem. Samotný objekt byl pak instancí konkrétní třídy. Smalltalkové objekty mohly vykonávat 3 základní věci: 1. Uchovávat stav (a reference na další objekty), 2. Přijímat zprávy samy od sebe nebo od jiných objektů, 3. V souvislosti se zpracováním zprávy odeslat zprávu samy sobě nebo jinému objektu.

Objevila se také verze podobná Smalltalku (kompilovaná Just In Time), která byla určena hlavně pro skriptování. Jejím autorem byl David Simmons a jmenovala se S#. Jazyky Python a dokonce i jazyk Ruby převzaly hodně z původního Smalltalku a přidaly ještě více C/Java syntaxe.

3.9. Scheme

Je multiparadigmový programovací jazyk, ale je znám především jako funkcionální (procedurální). Vytvořili ho Guy L. Steele a Gerald Jay Sussman v 70. letech 20. století. Dva standardy definovaly jazyk Scheme: Oficiální IEEE standard a druhý Revised Report on the Algorithmic Langage Scheme označovaný jako RnRS, kde n značí číslo revize. V současnosti je verze revize 5 a 6 je ve vývoji. Filozofií jazyka Scheme je minimalistická koncepce. Nabízí tak jednoduché pojmy a termíny, jak je to jen možné. Tam, kde je to praktické, nabízí vše ostatní pomocí programovacích knihoven. Je jedním z prvních jazyků, které podporovaly tzv. "continuations". Jedná se o zachování výkoného stavu programu (např. volání zásobníku a ukazatel na další instrukci). Tento stav se uloží a posléze se načte a tím dojde k pokračování samotného běhu programu.

Jazyk Scheme umožňuje také tzv. TAIL rekurzi. Jedná se o speciální případ rekurze, kdy se stává poslední operace funkce rekurzivním voláním. Nemusí se tak zapisovat místo návratu do zásobníku, protože daná funkce rekurzivně zavolá sama sebe a vrátí výsledek. Takto se dá značně odlehčit mnohdy přeplněnému zásobníku.

Kapitola 4. Programovací jazyk Ruby

4.1. Historie

Citován zdroj: [5]

Počátek vzniku jazyka Ruby sahá do roku 1993, přesně 24. února, kdy se Yukihiro Matsumoto rozhodl napsat si svůj plně objektově orientovaný skriptovací jazyk. V té době OO podpora v Perlu byla ve stádiu plánování a v Pythonu, podle názoru autora, nebyla plně integrována (některé objekty nejsou instancemi tříd). Ruby je napsán v jazyce C a je inspirován především Perlem, v některých částech je možné vysledovat podobnost s jazyky Eiffel a Ada. V roce 1995 vypouští první verzi, kolem projektu se formuje komunita a Ruby začíná být v Japonsku více populární než Python.

4.2. Filozofie Ruby - Bill Venners v rozhovoru s autorem Ruby

Přeloženo z anglického originálu: zdroj [7]

Yukihiro Matsumoto, autor programovacího jazyka Ruby, si povídá s Bilem Vennersem o designu a filozofii jazyka, o jeho nedokonalostech, přednostech a důležitosti člověka v počítačovém světě. Yukihiro Matsumoto, nebo "Matz," jak je známý online, je autorem programovacího jazyka Ruby. Ruby je objektově orientovaný jazyk pro psaní "day to day" skriptů a to jako plnohodnotných aplikací. Matz začal pracovat na Ruby před rokem 1993, protože chtěl jazyk, který mu pomůže být produktivnějším, zatímco bude zábavný a příjemný při použití. Ruby, zpočátku populární v Japonsku, si našlo cestu do srdcí programátorů po celém světě. 24 září 2003 se potkal Bill Venners s Yukihiro Matsumotem na JAOO konferenci v Aarhusu, v Dánsku. V tomto rozhovoru, který byl publikován na Artima.com, Yukihiro Matsumoto diskutuje o návrhu a filozofii Ruby, o znacích tohoto jazyka, a o tom, jak se stát lepším programátorem. Matz také řeší nebezpečí - problém ortogonality (vysvětleno pod odstavcem). Jeho jazyk poskytuje svobodu v principu co nejmenšího překvapení a zdůrazňuje roli člověka - programátora při vývoji.

Problém ortogonality - Ortogonalita nebo pravost, přímost je jedna z důležitých vlastností nejen aplikací. V pravém ortogonálním návrhu nemají operace - metody žádné vedlejší efekty. Neovlivňují jiné děje. Jsou přímo zodpovědné jen za jednu určitou činnost a nesmí ovlivnit činnosti ostatní. Můžeme uvést příklad z praxe. Na vašem monitoru u PC můžete měnit jas. Když změníte jas, změníte POUZE jas. Tato změna neovlivňuje třeba nastavení barev. Kdyby tomu tak bylo, museli byste při každé změně jasu ještě zkontrolovat a přenastavit barvy. Tyto dvě činnosti byste museli provádět současně a vyvažovat změny. Tak tomu je i v programování, kde jsou tyto děje nežádoucí. Proto je ortogonalita velice důležitým aspektem při vývoji aplikací.

Bill Venners: Dave Thomas, spoluautor Programming Ruby: A Pragmatic Programmer's Guide, mi řekl, že si nemyslíte o svém jazyce, že by byl navržen jako perfektní. Proč ne?

Yukihiro Matsumoto: Vývojáři programovacích jazyků chtějí perfektní jazyk. Chtějí říci, "Můj jazyk je perfektní. Dovede cokoliv." Ale je prostě nemožné vyvinout naprosto perfektní jazyk, protože existují dva různé pohledy na programovací jazyk. Jeden pohled se zabývá tím, co by se s jazykem mělo dělat - účelem, k jakému byl určen. Ten druhý říká, jak se budeme cítit při programování v tomto jazyce. Protože v Turingově kompletní teorii, každý Turingově kompletní jazyk může být teoreticky zastoupen jiným Turingově kompletním jazykem, ale za různou cenu. Můžete všechno tvořit v Assembleru, ale nikdo už nechce dnes v Assembleru programovat. Z úhlu pohledu - co můžete s jazykem dělat. Zde jsou jazyky pracující odlišně, ale rozdíly se stírají. Například Python a Ruby poskytují programátorovi tu samou sílu . Bez toho, aniž bychom zdůrazňovali účel, chci zdůraznit pohled JAK: Jak se budeme cítit při programování. To je hlavní odlišnost Ruby od ostatních jazyků. Zdůrazňuji pocit, zvláště to, jak se cítím při programování v Ruby. Nepracoval jsem tvrdě na tom, aby byl Ruby perfektní pro každého, protože každý z vás se cítí jinak. Žádný jazyk nemůže být perfektní pro všechny. Zkusil jsem vytvořit perfektní Ruby pro mě, ale možná není perfektní pro vás. Perfektní jazyk pro Guido van Rossuma je pravděpodobně Python.

Bill Venners: Dave Thomas také tvrdil, že když se vás zeptal na přidání znaku ortoganility, vy jste to nechtěl. To, co chcete je něco, co je harmonické. Co to znamená?

Yukihiro Matsumoto: Já věřím, že konzistence a ortogonalita jsou nástroje návrhu, ne primární cíl návrhu.

Bill Venners: Co znamená ortogonalita v tomto kontextu?

Yukihiro Matsumoto: Příklad ortogonality je povolování všech kombinací charakteristik nebo syntaxe. Například C++ podporuje oba výchozí parametry funkcí a přetěžuje názvy funkcí postavených na parametrech. Oba jsou dobré pro to mít je ve svém jazyce, ale proto, že jsou ortogonální, můžete je použít oba najednou. Kompilátor ví, jak je použít oba najednou. Pokud je to nejednoznačné, kompilátor vyhodí chybu. Ale když se podíváte na kód, Musíte uvažovat o pravidlech sami. Musím hádat, jak pracuje kompilátor. Pokud jsem dobrý a chytrý, není problém. Pokud ale nejsem dostatečně chytrý, a to opravdu nejsem, způsobuje to zmatek. Výsledek bude neočekávaný pro normální osobu. To je příklad, kdy je ortogonalita špatná.

Bill Venners: Jinými slovy, ortogonální znaky budou pracovat jedině v případě, pokud jim tvůrce kompilátoru porozumí a zprovozní je. Ale je to těžké pro programátory, protože je komplikované vyřešit jak tyto věci pracují současně.

Yukihiro Matsumoto: Pokud se kombinují ortogonální rysy, může se to rozvinout ve složitý problém.

Bill Venners: Takže, jaká je alternativa? Co by bylo více harmonické?

Yukihiro Matsumoto: Vzít do jazyka pouze jednu z těchto. Nemůžete dělat všechno, na co myslíte. Potřebujete jenom jednu z nich, i když jsou obě dobré.

Bill Venners: Jedna z návrhových filozofií komunity Pythonu je poskytování jedné a jediné cesty k řešení problému. Když poskytujete 50 různých cest, jak řešit to samé, pak poskytujete výhodu programátorům. Lidé mohou psát kód svou oblíbenou cestou. Nevýhoda platí pro ty, co kód čtou. Když čtu kód, mohl jste ho napsat jinak. Když čtu kód od jiné osoby, mohl to napsat

zase po svém. Takže jako čtenář kódu musím být schopen číst všechny možné způsoby zápisu, a to i přesto, že to není můj oblíbený styl zápisu. To je návrhový kompromis. Komunita Pythonu preferuje jen jednu možnou cestu, ale Ruby poskytuje několikanásobnou cestu k řešení problému.

Yukihiro Matsumoto: Ruby zdědilo od Perlu filozifii více cest k řešení problému. Odvodil jsem tuto filozofii od Larryho Walla, který je aktuálně můj hrdina. Chci dát uživatelům Ruby volnost. Chci jim dát svobodu volby. Lidé jsou rozdílní. Vybírají si podle různých kritérií. Ale pokud je zde lepší cesta oproti spoustě alternativ, chci prozkoumat tuto cestu a problém vyřešit elegantně. To je to, o co jsem se pokoušel. Možná, že kód Pythonu je o něco lépe čitelnější. Každý může psát stejným stylem Pythonu, a může to být mnohem čitelnější, možná. Ale rozdíly mezi jednotlivými programátory jsou tak velké, že poskytování jen jedné cesty u Pythonu pomáhá mnohem méně. Myslím, raději poskytovat tolik možností, kolik jen je možné, ale doporučit uživatelům tu nejlepší cestu.

Bill Venners: V úvodu článku o Ruby píšete "Pro mě je účel života mít z něj radost. Programátoři často cítí radost, když se soustředí na vlastní kreativitu při programování, Takže Ruby je vytvořeno k tomu, aby byli programátoři šťastní." Jak může Ruby dělat programátory šťastnými?

Yukihiro Matsumoto: Chcete si přece užít život nebo ne? Pokud uděláte práci rychle a ještě je zábavná, je to dobře, nemyslíte? To je účel života. Váš život je lepší. Chci řešit problémy, které potkávám v běžném životě pomocí počítačů, takže potřebuji psát programy. Používáním Ruby se můžu soustředit na věci, které dělám. Žádná magická pravidla v jazyce jako PUBLIC VOID něco něco něco, k tomu říci "print hello world." Chci pouze říci, "print this!" Nechci nikoho obklopeného magickými klávesovými zkratkami. Chci se pouze soustředit na daný problém. To je základní myšlenka. Takže jsem se snažil vytvořit kód Ruby stručný a strohý.

Bill Venners: Povolení programátorům psát kratší a úspornější kódy je cesta k jejich radosti.

Yukihiro Matsumoto: Ano, tak se mohou soustředit na vlastní problém. Někdy lidé píší pseudokód na papír. Když tento pseudokód běží rovnou na jejich počítači, je to to nejlepší, nebo ne? Ruby se snaží být takové, jako pseudokód, který běží. Lidé od Pythonu to říkají také.

Bill Venners: Ano, lidé od Pythonu říkají, že Python je spustitelný pseudokód. Co dalšího v Ruby dělá programátory šťastnými?

Yukihiro Matsumoto: V našich každodenních životech programátorů, zpracováváme mnoho textových řetězců. Snažil jsem se tvrdě pracovat na zpracovávání textu, jmenovitě na třídě STRING a na regulárních výrazech. Regulární výrazy jsou zabudovány v jazyce a jsou velmi propracované. Hodně často také potřebujeme volat události a programy v operačním systému. Ruby umí volání v jakémkoliv UNIXovém stroji a v převážné většině Windows API. To přináší sílu a funkcionalitu do interpretovaného jazyka. Takže můžete dělat denně systémovou administraci a programovat zpracování textu. To je doménou mých programátorských aktivit, takže jsem tvrdě pracoval právě na tomto.

Bill Venners: Takže v základu mi Ruby pomáhá užívat si život naplno a dělat svoji práci mnohem rychleji a přitom se zábavou?

Yukihiro Matsumoto: Pomáhá mi právě v tomhle. nejsem si jistý, že to dělá pro vás, ale doufám, že ano.

Bill Venners: V rozhovoru říkáte, "Nepodceňujte lidský faktor. Dokonce, že jsme před počítači, které jsou našimi médii. Pracujeme pro lidi a s lidmi." Co tím myslíte?

Yukihiro Matsumoto: Představte si, že píšete email. Sedíte před počítačem. Ovládáte počítač klikáním na myš a psaním na klávesnici, ale zpráva bude odeslána přes internet konkrétnímu člověku. Takže vy pracujete před počítačem, ale člověk za počítačem. Většinu úloh tvoříme pro lidi. Například výpočet daně je počítání čísel, kterým může vláda dostat peníze z mojí pěněženky, ale vláda obsahuje lidi. Většina úkolů je spojena s lidmi. V programování tedy chceme, aby počítač pracoval pro člověka, nebo se snažíme popsat počítači naše úvahy a postupy a to velmi jednoznačnou cestou, kterou může počítač vykonat. V prvním případě chceme, aby počítač pracoval pro člověka a cílem je další člověk za počítačem. Ve druhém případě sdělujeme naše myšlenky počítačům, které je vykonávají, předkládáme podnět z našich mozků a výsledek je vytvořen počítačem. V obou případech zde vystupuje člověk.

Bill Venners: Co je nejdůležitější v této myšlence? říkáte, "Nepodceňujte lidský faktor." Proč?

Yukihiro Matsumoto: Protože počítače nemyslí. Když musím vytvořit pracný způsob komunikace s počítačem nebo když je snadné s ním komunikovat. Počítače se nestarají. Když zadám čísla nebo jednotlivé instrukce a spustím je, nebo nebo když vysokoúrovňový jazyk vygeneruje instrukci. Počítače se o nic nestarají. My, lidé se staráme o práci, za kterou nás platí. Často lidé, počítačoví inženýři, hledí na své stroje. Myslí si, "Kdyby se udělalo tohle nebo támhleto, stroj by byl rychlejší. Pomocí tohoto bude stroj mnohem efektivnější, atd." Zaměřují se na stroje. Ale my se potřebujeme zaměřit na lidi, na to, jak se lidé starají, jak programují, jak obsluhují uživatelské aplikace. My jsme pány. Oni jsou sluhové.

Bill Venners: Prozatím tomu tak je.

Yukihiro Matsumoto: Prozatím, dokud nepřijde čas Terminátora.

Bill Venners: V rozhovoru jste řekl "Vytvořil jsem Ruby, abych minimalizoval svá překvapení. Byl jsem ohromen tím, že ostatní lidé po celém světě říkají, že Ruby redukovalo jejich překvapení a zvětšilo jejich radost při programování. Teď jsem si jistý, že programátorské mysli jsou si podobné na celém světě." Proč princip snižování překvapení?

Yukihiro Matsumoto: Ve skutečnosti jsem neprohlašoval, že by Ruby následovalo filozofii snižování překvapení. Někteří lidé to tak mohou cítit a tak to začali říkat. Ve skutečnosti jsem to neřekl já. Chtěl jsem zmenšit mou frustraci při programování, tak, že jsem chtěl minimalizovat své úsilí. To byl můj hlavní cíl při vývoji Ruby. Chtěl jsem mít z programování radost. Po vypuštění Ruby a když o něm začali lidé ve světě vědět, říkali mi, že cítili, jak jsem se cítil já. Oni přišli s frází menšího překvapení. Ale ve skutečnosti je to často nepochopeno.

Bill Venners: Jak nepochopeno?

Yukihiro Matsumoto: Každý má jiný individuální základ. Někdo pochází z komunity Pythonu, někdo jiný začínal s Perlem, a mohou být překvapeni rozdílnými aspekty těchto jazyků. Pak přicházejí za mnou a říkají, "byl jsem překvapen tímto rysem jazyka." Počkat, počkat. Princip snižování překvapení není pouze pro vás. Tento princip znamená můj pocit. Tento princip poznáte, až se naučíte dobře Ruby. Například já byl předtím C++ programátor. Začal jsem s Ruby.

Programoval jsem v C++ tři, čtyři roky. I po dvou letech jsem mohl říci, že mě C++ pořád překvapuje.

4.3. Charakteristické rysy jazyka Ruby

Napsán v jazyce C

Skriptovací jazyk Ruby je napsán v programovacím jazyce C (tedy jeho interpret), což mu skýtá i jisté výhody. Jednak co se týče rychlosti ale i kompatibility pro různé platformy.

Modulární jazyk

Volně čerpáno z: zdroj [19]

Kromě tříd a dělení kódu do souborů můžeme v Ruby použít i tzv. **MODUL**. Jedná se o část kódu velmi podobnou třídě. Narozdíl od souboru, který vkládáme do kódu příkazem require, modul připojujeme pomocí include. Vkládáním modulů do definic tříd vznikají tzv. **MIXINY**. Převážná část standardní knihovny Ruby je organizována do modulů (s výjimkou tříd).

MODUL:

```
module Matphys

  def distance(x1,y1,x2,y2)
    Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))
  end

end
```

MIXIN:

```
class Object

  # modul se použije v rámci definice třídy
  include Matphys

end
```

Použití:

```
# voláme metodu, která přibyla do třídy 'Object'
puts distance(3,4,5,6)
```

Dynamické typování

- (někdy používaný výraz **polymorfismus proměnných**)

- v anglickém programátorském slangu se také užívá výraz: **DUCK TYPING**

"If it walks like a duck and quacks like a duck, then it must be a duck."

citován zdroj [18]

Jazyk Ruby používá standardní typy podobné jako u kompilovaných jazyků. Např:

String ..... Reprezentace znakového řetězce, spousta dostupných metod pro převody a transformace stringů.

Integer ..... Reprezentace hodnoty celočíselného významu.

Float ..... Reprezentace reálných čísel.

Array ..... Reprezentace spojového seznamu, ale i libovolné datové spojové struktury

Hash ..... Reprezentace seznamu dvojic (klíč, hodnota).

Regexp ..... Regulární výrazy (pro parsování textu - stringů).

a další typy ...

Poznámka

Ruby zachází s typem Integer následujícím způsobem. Dodržuje filosofii, že všechno musí být objekt, tedy i tyto primitivní typy jsou zabaleny do objektu. Ruby používá pro integery dva obalové typy. **Fixnum** *pro "malá" čísla a* **Bignum** *pro "velká". Programátor se v podstatě nestará o to, jaký typ je zrovna použit, Ruby samo obaluje na Fixnum nebo Bignum. Konverze je automatická.*

Používání typů proměnných v názorné ukázce:

```
prom=Array.new
prom2="Ahoj světe"
```

Zavedli jsme dvě proměnné **prom** a **prom2**. Před touto definicí jsme jasně Ruby neřekli jakého typu mají tyto proměnné být. Mohli jsme je použít a používat jakkoliv. Na těchto dvou řádcích dáváme Ruby jasně na srozuměnou, že **prom** budeme používat jako **datovou strukturu spojový seznam** a **prom2** jako **stringovou proměnnou**, do které jsme hned vložili řetězec "Ahoj světe".

Jako důkaz, že proměnné můžeme používat střídavě, jak se nám zrovna zlíbí (i když tento postup bych zrovna nedoporučoval, protože se pak v typech proměnných snadno ztratíte) uvedu tento příklad:

```
prom=0
print prom+1
prom="Hello world"
print "\n"+prom
```

Program vypíše následující (ve skriktně typovaných jazycích nemyslitelné):

```
1

Hello world
```

Označení proměnných

bez označení ..... lokální proměnná

@ ..... instanční proměnná

$ ..... globální proměnná

1. velké písmeno ..... konstanty

```
tmp
@name
$glob_name
```

Proměnná **tmp** je klasická lokální proměnná. Její platnost je definována pouze pro daný blok, pro metodu, atd.

Proměnná **@name** je instanční proměnná, tedy vlastnost konkrétní třídy. Její platnost je samozřejmě v celé třídě.

Proměnná **$glob_name** je globální proměnná.

Linecne GPL

Ruby je šířeno pod licencí GNU GPL. Tedy příjemci softwaru jsou poskytnuta stejná práva jako má autor programu, nebo jaká byla poskytnuta autorovi právě licencí GPL. Takovýto svobodný kód tedy nesmí být využit v proprietárním softwaru.

Poznámka

Proprietární software je software, který není ani svobodný, ani částečně svobodný. Jeho svobodné užívání, změna či šíření jsou zakázány nebo musíte žádat o povolení; případně jsou omezení taková, že to vlastně nemůžete svobodně dělat. (-- volná definice proprietárního softwaru --)

http://www.gnu.no/philosophy/categories.cs.html (Nadace pro svobodný software)

4.4. Praktické rysy jazyka Ruby - vlastní výzkum

Více možností zápisu - KOLEKCE

Sám autor vychvaluje jazyk Ruby pro své možnosti zapsat konkrétní věc ne dvěma, nebo třemi ale hned několika způsoby. Na vlastní kůži jsem se přesvědčil, že to není jen reklama. Ruby opravdu vychází v tomto směru vstříc všem programátorům. Než jsem si osvojil filozofii Ruby,

chvíli mi to trvalo. Snažil jsem se za každou cenu najít správnou syntaxi, jak se daný problém má zapsat. Časem jsem přišel na to, že na to logicky mohu přijít zcela sám. Zápis je intuitivní, pokud si osvojíte pár základních jednoduchých pravidel. Například takové používání kolekcí. V Javě nebo v C# se musí přesně dodržet konvecence zápisu, v podstatě existuje jediná správná možnost. Ono všechno má své klady i zápory. Z hlediska programátora je ale Ruby mnohem pohodlnější. Když neznáte, nebo tápete, stačí zapsat problém podobně, jak jste zvyklí z Javy nebo z Céčka. Chyba to není, program funguje. (Samozřejmě rozdíly tu jsou, o definování typů se rozepíši dále) Z hlediska efektivity kódu a jednoduchosti zápisu se dá použít i zcela jednoduchá možnost, která vám předchozí zápis silně zredukuje. A tím zdaleka možnosti Ruby nekončí. Zůstaneme u příkladu kolekcí. Pokud potřebujeme vytvořit kolekci, stačí použít konstruktor:

Array.new

Na všeobecná pole, arrayListy, seznamy, atd.

Pokud chceme použít speciální případ struktury HASH tabulku, stačí použít třídu Hash. Konstruktor **Hash.new**

Tím končí všechny konvence a my se můžeme soustředit přímo na daný úkol. Podle toho, jak chceme se strukturou pracovat, tak se k ní chováme, tak zapisujeme jednotlivé operace s ní. Při vkládání prvku, při výběru nalezneme v dokumentaci Ruby spoustu metod, ale můžeme použít v podstatě cokoliv, co nás napadne. Třeba pro průchod seznamem můžeme použít intuitivní metodu each. Nic nám ale nezakazuje použít vlastní iterátor a projít seznam pomocí cyklu. Použití záleží na programátorovi, ne na dané syntaxi. Máme obrovskou možnost volby. Při zjišťování obsahu kolekce nabízí Ruby asi nejvíc intuitivních metod oproti jiným jazykům. Jsou tu připravené metody pro výpis celé kolekce , můžeme si i určit jak se přesně má co vypsat, což se dá prakticky zapsat do tzv. bloků, které jsou uzavřené složenými závorkami {}. Právě tyto bloky se dají použít pro metodu each, kdy uzavřeme do bloku prováděný kód pro každý prvek. Jedná se o nahrazení cyklu. Ve strojovém kódu se tedy stejně jedná o cyklus pro průchod, ale zápis pro programátora je mnohem jednodušší a rychlejší (z hlediska psaní kódu). Metodu each najdeme ve všech kolekcích. Pozoruhodnou možnost nabízí i použití tohoto bloku při řazení prvků kolekce. Když jsem objevil tento jednoduchý a jednořádkový způsob zápisu, začal jsem mít z programování v Ruby opravdu radost. V Javě se tento problém řeší úplně jinak, standardně a pokud chceme řadit prvky v kolekci podle určitého pravidla, musíme zapsat celý blok kódu, který nám překryje standardní řazení prvků. U Ruby stačí zapsat zkratku. Symboly dvou prvků a co s nimi chceme udělat, dalo by se říci dva prvky a operátor za použití bloku {}. Dostaneme se na jeden řádek kódu.

Java:

V Javě pro řazení prvků kolekce používáme buď rozhraní Comparable nebo Comparator. V prvním případě musíme implementovat rozhraní Comparable pro jednotlivé prvky a překrýt metodu compareTo(), která určuje, podle jakých pravidel se budou dva prvky navzájem porovnávat. V případě použití Comparatoru si můžeme vytvořit přímo vlastní Comparator, který pak použijeme při konstrukci kolekce jako parametr, atd. Zde musíme překrýt metody compare a equals. Tato Javovská konvence nám zaručuje bezproblémový chod, přehled a jednoznačnost. Bohužel je to psaní obsáhlejšího úseku kódu, překrytí metod, implementace rozhraní. Programátor tu stráví přece jen hodně času nad samotnou konstrukcí aniž by řešil problém.

Ruby:

Ruby se spokojí v nejjednoduším možném případě pouze s jednou řádkou kódu. Jedná se o zkratku, tedy nedá se z ní zcela jednoznačně vyčíst její účel, ale programátorovi poskytuje značný komfort. Pokud máme naplněnou kolekci čísly a chceme tato čísla seřadit od nejmenšího po největší. Stačí použít metodu sort u libovolné kolekce. Ruby rozlišuje ještě **sort** a **sort!**. Metoda sort vrací jinou kolekci setříděnou podle pravidel, která následují. Metoda sort! přeskupí prvky přímo v kolekci, ze které je volána. Tedy častější použití bude metody sort!. Do bloku {} za metodu napíšeme pravidla, podle kterých se budou prvky porovnávat. Pro porovnání čísel stačí tedy napsat:

```
kolekce.sort! {|a,b| a<=>b}
```

přičemž proměnné v || nám symbolizují ony dva porovnávané prvky, následuje porovnávací pravidlo.

S kolekcemi se dá v Ruby dělat obecně spousta věcí přímo. Je připraveno nepřeberně metod, které vykonávají různé obvyklé ale i méně obvyklé operace nad kolekcemi. Tyto v ostatních jazycích tak trochu chybí, nebo spíš jsou natolik doplňující, že si je programátor většinou vytváří sám při implementaci. Neubráním se uvedení pár příkladů, které potěší oko programátora svou jednoduchostí a přímostí.

Velice zajímavá možnost je porovnávání přímo celých naplněných kolekcí. Jednoduchá otázka na dvě kolekce, bez zbytečných podrobností. Samozřejmě si musíme i v těchto případech dávat pozor na speciální řazení prvků, ale pokud nechceme po Ruby nic speciálního, vyplatí se jistě tato jednoduchá forma dotazu. Pro porovnání dvou kolekcí stačí tedy napsat:

```
[ "a", "a", "c" ] <=> [ "a", "b", "c" ]
```

V tomto případě se hodnota výrazu rovná -1, protože první kolekce je menší než druhá, z pohledu abecedního řazení prvků.

Následující případ se už rovná jedničce, zde zapsaný vztah platí (první kolekce je větší než druhá):

```
[ 1, 2, 3, 4, 5, 6 ] <=> [ 1, 2 ]
```

V případě rovnosti kolekcí tento dotaz vrací 0. Můžeme se také zeptat přímo pomocí operátoru ==

```
[ "a", "c", 7 ] == [ "a", "c", 7 ]
```

Tento operátor vrací logickou hodnotu TRUE/FALSE.

Dále můžeme v kolekcích přímo vyhledávat jednotlivé stringové vzory, kolekce různě slučovat, třídit, rozdělovat. Vše jednoduše pomocí připravených metod. Můžeme mazat prvky z kolekce za splnění určité podmínky, což se zapisuje zase do bloku { |a| a<"b" }. Zde konkrétně tento blok v metodě *delete_if* by mazal pouze prvky, které jsou abecedně menší než "b". Tady už se vyplatí pohled do dokumentace, jelikož tyto metody by běžný programátor ani neočekával. I

podle samotného autora Ruby se ale právě s těmito operacemi nad kolekcemi a kolekcemi stringů setkáváme ve většině aplikací.

Jako velmi užitečná metoda se ukázala metoda *inspect*, která vrací string:

```
pole=[1,2,3,4,5]
print pole.inspect

[1,2,3,4,5]
```

Tato metoda dokáže přijatelně vytisknout celou kolekci v rozumném formátu. Tedy pokud potřebujeme jisté ladící výpisy kolekcí, nebo prostý výstup kolekce, tento výstupní string je velice šikovný. K úpravě tohoto stringu slouží metoda *join*, u které si můžeme ještě nadefinovat separátor - oddělovač, kterým budou ve výpisu jednotlivé prvky oddělené. U těchto výpisů kolekce je možno použít více metod, které jsou zodpovědné všechny za to samé. Obdobně lze použít i metodu *to_s*, kterou je známá spíše z ostatních skriptovacích jazyků. Najdeme zde spoustu alternativ, takže i programátor, který nikdy neviděl dokumentaci a nezná konkrétní metody, se může snadno chytit a začít psát kód.

Ruby v kolekcích počítá i s jejich přímým ukládáním do souboru. Zde se uplatní i metoda *pack*, která dokáže kolekci zabalit do binárního stringu a to pomocí námi definované direktivy. Celou kolekci tak můžeme převést v podstatě na cokoliv, na ASCII znaky, na hexa čísla, atd. Pro rozbalení kolekce existuje samozřejmě metoda *unpack*.

Praktická ukázka práce s kolekcemi - ukázkový program

Pro názornější přiblížení práce s kolekcemi jsem se rozhodl vytvořit malý ukázkový program. Jedná se o abstrakci univerzity, která sdružuje akademickou obec, tedy studenty a učitele. Samozřejmě, že univerzita má akreditované předměty, které navštěvují studenti a vyučují je daní učitelé. Pro tyto objekty zavádíme třídy Ucitel, Predmet, Student, Univerzita a speciální testovací třídu Test, která otestuje funkčnost programu na konkrétních datech. Univerzita tedy uchovává seznam učitelů, seznam předmětů i studentů. Do těchto kolekcí lze vkládat, tedy přidávat další údaje. V rámci třídy Univerzita budeme chtít ale znát odpovědi i na další otázky. Například jaké předměty učí konkrétní učitel nebo seřazený žebříček studentů podle prospěchu - váženého průměru.

Třída Student:

Uchovává v první řadě jméno studenta, jeho příjmení a identifikační číslo. Dále pak seznam předmětů (ukazatele na instance třídy Predmet), které má daný student zapsané. Student se může zapsat na předmět nebo snadno zjistit jaké předměty má zapsány.

```
require 'predmet.rb'

#Trida reprezentujici studenta na univerzite
class Student

  ...

  #metody:
```

```
  #Konstruktor
  def initialize(jmeno,prijmeni,cislo)
    @jmeno=jmeno
    @prijmeni=prijmeni
    @cislo=cislo
    @predmety=Array.new
  end

  def initialize(jmeno,prijmeni,cislo,predmety)
    @jmeno=jmeno
    @prijmeni=prijmeni
    @cislo=cislo
    @predmety=predmety
  end

  ...

  #Studuje predmet?
  def studujePredmet(predmet)
    if(@predmety.include?(predmet))
      return true
    else
      return false
    end
  end

  #Prihlas se na predmet
  def prihlasNaPredmet(predmet)
    @predmety.push(predmet)
  end
end
```

Vidíme zde použití metody *include?*, pomocí které se ptáme na předměty, které má daný student zapsány. Dále pro vkládání dalších předmětů do kolekce je použita metoda *push*. Je také patrné, že třída používá třídu Predmet, která je tedy k této třídě přidána pomocí *require*.

Třída Ucitel:

Třída Učitel nám pouze uchovává základní informace o jednotlivých učitelích, nic světoborného. Samotná třída ani nepracuje s kolekcemi. Upozorním tedy alespoň na techniku psaní instančních proměnných a accessorů k nim. Použil jsem zažitý Javovský přístup, ale lze použít i jiné postupy a mnohem rychlejší, jak ukáži v kódu níže, kde přepíši třídu Ucitel pomocí doporučeného zkráceného zápisu v Ruby, které vytvoří accessory k instančním proměnným samo. Programátor jen určí, jestli máme do proměnné přístup pouze pro čtení nebo zápis a nebo pro obojí.

```
class Ucitel

  #instancni promenne:
```

```
  #titul
  def titul
    @titul
  end

  #jmeno
  def jmeno
    @jmeno
  end

  #prijmeni
  def prijmeni
    @prijmeni
  end

  #Metody
  #Konstruktor:
  def initialize(titul,jmeno,prijmeni)
    @titul=titul
    @jmeno=jmeno
    @prijmeni=prijmeni
  end

  #vrat ucitele
  def getUcitel()
    return @titul+" "+@jmeno+" "+@prijmeni
  end
end
```

!!! V Ruby se převážně používá jiný způsob zápisu instančních proměnných:

```
class Ucitel

  #instancni promenne:

  #titul
  attr_accessor: titul

  #jmeno
  attr_reader: jmeno

  #prijmeni
  attr_reader: prijmeni

  #Metody
  #Konstruktor:
  def initialize(titul,jmeno,prijmeni)
    @titul=titul
```

```
    @jmeno=jmeno
    @prijmeni=prijmeni
  end

  #vrat ucitele
  def getUcitel()
    return this.titul+" "+this.jmeno+" "+this.prijmeni
  end
end
```

attr_accessor: titul nám vytvoří accessor (přístupová práva pro čtení i zápis do proměnné), metody pro přístup k proměnné titul. K zapisování stačí přistoupit přes tečku k objektu a za jméno proměnné dát rovnítko a naplnit jí novou hodnotou. Pro čtení stačí přes tečku přistoupit k proměnné, vrátí se její hodnota.

attr_reader vytváří pouze metodu (přístup) pro čtení.

attr_writer pouze pro zápis do proměnné.

Poznámka

Samozřejmě fungují v Ruby i všeobecně známé modifikátory přístupu. Tedy **public**, **protected** *a* **private***, pro veřejný, chráněný a soukromý mód. Ve veřejném módu je položka přístupná všem, v chráněném módu je přístupná pouze z objektu téže třídy nebo třídy od ní odvozené (oděděné), soukromý mód umožňuje pak přístup jen jednomu konkrétnímu objektu (instanci).*

Třída Predmet:

Třída Predmet abstrahuje informace o jednotlivých předmětech, tedy jejich název, odkaz na učitele, který předmět vyučuje a počet kreditů za splnění předmětu. Dále pak obsahuje HASH strukturu výsledky, do které se zapisují výsledky všech studentů, kteří mají zapsaný předmět. Výsledky se zapisují ve tvaru klíče a hodnoty, kde klíčem je identifikační číslo studenta a hodnotou je známka. Třída je znovu jako všechny ostatní zapsána javovskou konvencí přístupu k instančním proměnným.

```
require 'ucitel.rb'

class Predmet

  ...

  #Metody:
  #Konstruktor
  def initialize(nazev,ucitel,kredity)
    @nazev=nazev
    @ucitel=ucitel
    @kredity=kredity
    @vysledky=Hash.new
```

```
  end

  ...
end
```

Můžeme si povšimnout vytvoření prázdné HASH struktury pro uložení výsledků v KONSTRUKTORU. Vytvoření proběhne pomocí *Hash.new*.

Třída Univerzita:

Obsahuje seznamy studentů, učitelů i předmětů, dále metody na zjištění "co učí který učitel" a hlavně metodu pro seřazení žebříčku studentů podle váženého průměru. Jako pomocné metody najdeme jednu pro zjištění známky studenta z konkrétního předmětu a dále potom metodu pro spočítání váženého průměru.

```
require 'student.rb'
require 'predmet.rb'
require 'ucitel.rb'

class Univerzita

  #instancni promenne:

  ...

  #Metody
  #Konstruktor:
  def initialize(nazev,mesto)
    @nazev=nazev
    @mesto=mesto
    @studenti=Array.new
    @ucitele=Array.new
    @predmety=Array.new
  end

  ...

  #Vypis seznam studentu
  def getStudents()
    vrat=""
    @studenti.each do |student|
      vrat+=student.getJmeno()+"\n"
    end
    return vrat
  end

  #Vypis seznam predmetu
  def getPredmets()
    vrat=""
```

```
  @predmety.each do |predmet|
    vrat+=predmet.popis()+"\n"
  end
  return vrat
end

#Vypis seznam ucitelu
def getUcitels()
  vrat=""
  @ucitele.each do |ucitel|
    ucitel.getUcitel()+"\n"
  end
  return vrat
end

#Co uci konkretni ucitel
def coUci(ucitel)
  predmety=Array.new
  @predmety.each do |predmet|
    if(predmet.getUcitel()==ucitel)
      predmety.push(predmet)
    end
  end
  return predmety
end

#Zebricek studentu celkove - od nejlepsiho po nejhorsiho
def getZebricek()
  zebricek=Hash.new
  @studenti.each do |student|
    znamky=Hash.new
    student.getPredmety().each do |predmet|
      znamky[predmet.getKredity()]
      =najdiZnamkuStudenta(student,predmet)
    end
    zebricek[student.getJmeno()]=vazenyPrumer(znamky)
  end
  return zebricek.sort {|a,b| a[1]<=>b[1]}
end

#Najde znamku ve vysledcich konkretnimu studentovi
def najdiZnamkuStudenta(student,predmet)
  predmet.getVysledky().each_pair do |cislo,znamka|
    if(student.getCislo()==cislo)
      return znamka
    end
  end
end
```

```
  #Pomocna metoda pro pocitani vazeneho prumeru studenta
  def vazenyPrumer(znamky)
    soucet=0
    soucetKreditu=0
    znamky.each_pair do |kredity,znamka|
      soucet+=(kredity * znamka)
      soucetKreditu+=kredity
    end
    return (soucet.to_f() / soucetKreditu.to_f())
  end
end
```

Při zjišťování, co učí konkrétní učitel, je použita klasická konstrukce pro procházení kolekce pomocí metody *each do*:

```
@predmety.each do |predmet|
      if(predmet.getUcitel()==ucitel)
        predmety.push(predmet)
      end
```

V metodě pro sestavení žebříčku studentů je použita speciální "zkratka" pro řazení podle váženého průměru:

```
zebricek.sort {|a,b| a[1]<=>b[1]}
```

Porovnávají se prvky HASH struktury žebříček a to ne podle klíče, ale podle hodnot.

Ještě jedna maličká zajímavost by se našla, v programu sice nepoužívaná metoda pro přidání studentů - druhá možnost:

```
@studenti+=studenti
```

V tomto případě se používá zkráceného zápisu práce se dvěma kolekcemi, kdy se kolekce studenti vloží do instanční proměnné (kolekce @studenti).

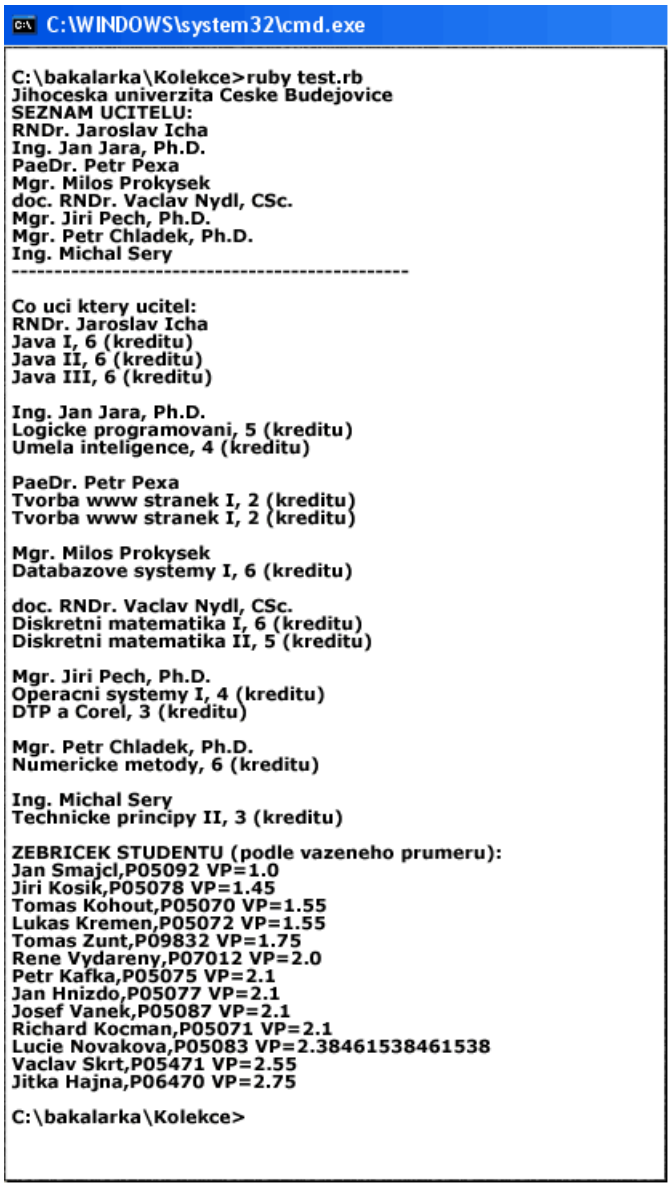
Testovací třída Test (Pro vyzkoušení funkčnosti programu):

Tato třída vytváří konkrétní objekty - instance tříd a naplňuje je daty. Vytvoří tedy univerzitu, přidá učitele, přijme studenty, ti se přihlásí na předměty. Učitelé vyplní hodnocení - výsledky jednotlivých studentů. V rámci běhu testovacího programu (třída Test) se vypíše nejprve seznam učitelů dané univerzity, poté seznam učitelů a co který učitel učí a nakonec se vypíše seřazený seznam studentů podle váženého průměru.

Výpis testování:

Spustíme třídu *Test* ze souboru **test.rb**:

Obrázek 4.1. Výpis testování



Kompletní zdrojové kódy k ilustračnímu příkladu jsou přiloženy na CD v adresáři Kolekce.

Zásobník nebo Fronta?

Jak jsem zmínil v úvodu, Ruby zachází s kolekcemi tak, jak zrovna chcete, jak potřebujete. Tedy pokud chcete vytvořit strukturu zásobníku nebo fronty, je to v podstatě úplně jedno. Vytvoříte obecnou kolekci a k té se pak chováte jako k zásobníku nebo jako k frontě. Záleží na způsobu (tedy metodách), který používáte při plnění a výběru. Pro zásobník se přímo nabízí intuitivní metody ***push*** a ***pop***. Pro frontu je pak třeba použít výběr prvního prvku, tedy metodu ***shift*** a pro vložení na konec fronty metodu **push**. Jinak se ale nemusíme starat o strukturu jako takovou. Zásobník a fronta se od sebe liší v podstatě jen způsobem výběru prvku. U zásobníku (paměť typu LIFO) se jako první vybere naposledy vložený prvek. U fronty (paměť typu FIFO) se jako první vybere prvek, který byl vložen jako první. Odebírá se tedy z čela fronty. Způsob vkládání prvků do struktury se nemění. Pokud bychom ale potřebovali vložit prvek na začátek struktury (na čelo fronty), stačí využít metodu **unshift**, která prvek zařadí na začátek.

Malý příklad zacházení s typem **Array** jako se zásobníkem a frontou:

```
zasobnik=Array.new
  zasobnik.push(1)
  zasobnik.push(2)
  zasobnik.push(3)
  zasobnik.push(4)
  zasobnik.push(5)
  print "Zasobnik:\n"+zasobnik.inspect
  print "\nVyber prvku z vrcholu zasobniku:\n"
  print zasobnik.pop
```

```
Zasobnik:
[1, 2, 3, 4, 5]
Vyber prvku z vrcholu zasobniku:
5
```

Vytvoříme proměnnou **zásobník** a definujeme ho prázdným spojovým seznamem - **Array**. Poté začneme postupně vkládat do "zásobníku" čísla od 1 do 5. Vytiskneme zásobník pomocí metody *inspect* a vybereme vrchol zásobníku pomocí metody *pop*. Dostaneme tedy v případě zásobníku číslo 5. Zásobník se tedy jako zásobník pouze tváří pomocí metod *push* a *pop*.

```
fronta=Array.new
  fronta.push(1)
  fronta.push(2)
  fronta.push(3)
  fronta.push(4)
  fronta.push(5)
  print "Fronta:\n"+fronta.inspect
  print "\nVyber prvku z cela fronty:\n"
  print fronta.shift
```

```
Fronta:
[1, 2, 3, 4, 5]
Vyber prvku z cela fronty:
1
```

Znovu vytvoříme proměnnou, tentokrát s názvem **fronta**, a definujeme ji prázdným spojovým seznamem - **Array**. Postupně vkládáme čísla od 1 do 5. Potud se nic nemění. V případě výběru prvky z fronty ale použijeme metodu **shift**, která nám vrátí první prvek a také ho z čela fronty odebere.

Je vidět, že nepotřebujeme definovat žádnou datovou strukturu typu Fronta nebo Zásobník. Pohodlně si vystačíme s dobře napsanou třídou Array. Tento postup by se dal označit jako další polymorfismus. Tentokrát ne polymorfismus proměnných ale přímo datové struktury.

Co se týká kolekcí, je Ruby vybaveno opravdu dokonale. Programátor se může věnovat spíše konkrétnímu problému než realizaci vlastní datové struktury. Samozřejmě za každé pohodlí se platí, někdy nečitelností kódu, někdy rychlostí aplikace. Je ale na každém jakou cestu si v Ruby

vybere a zde se přímo nabízí použití předem připravených optimalizovaných metod. Tedy jak v úvodu tvrdil sám autor, Ruby nabízí mnoho možností a preferuje tu nejefektivnější.

Více možností zápisu - CYKLY

V Ruby najdeme opravdu mnoho možností jak zapsat cyklus. Jedná se o kombinaci zápisů z různých jazyků, takže programátor může psát tak, jak je zvyklý. Zápisy cyklů jsou rovnocené, takže je opravdu jedno, jaký si vybereme.

Obyčejný cyklus **FOR** můžeme například zapsat:

```
for x in 0..100
end;
```

Zde se inkrementuje proměnná x od 0 do 100 včetně. Jedná se o jeden z nejkratších zápisů cyklu. Stejně tak můžeme procházet i seznam, tedy kolekci, kde za **in** napíšeme přímo kolekci.

Pro celočíselné typy můžeme poukžít i jednoduchou metodu *times*, jejíž použití je nanejvýš intuitivní. Například:

```
3.times { print "Ahoj" }
```

Třikrát vypíše slovo Ahoj.

Obdobně fungují metody *upto* a *downto*:

```
3.upto(7) { |x| print x }
```

Vypíše postupně čísla od 3 do 7. Metoda downto by fungovala sestupně.

Pokud chceme inkrementovat nebo dekrementovat o libovolný krok, můžeme použít metodu *step*:

```
10.step(0,-2) { |x| print x }
```

Cyklus začne na 10 a bude v každém průchodu odečítat 2, dokud se nedostane na 0. V následujícím bloku pak můžeme specifikovat, co se má vykonat. V našem případě vypsat hodnotu proměnné cyklu.

U procházení kolekcí stačí použít metoda *each* s vykonávacím blokem ve složených závorkách. Jedná se o nejsnadnější zápis průchodu kolekcí. Trochu více zajímavé je to, že se metoda *each* dá použít i pro průchod jednotlivými řetězci. Stringy můžeme procházet buď po řádcích nebo je dělit pomocí libovolných oddělovačů, které specifikujeme v metodě *each*. Můžeme string procházet i po znacích, které jsou reprezentovány ASCII kódem. Ten můžeme převést znovu na znak metodou *chr*.

```
retezec.each { |radek| print "- ", radek }
```

Tento příklad nám rozdělí řetězec po řádcích - tedy hledá znak **\n** a přidá před tento každý řádek znak -.

```
retezec.each_byte { |znak| print znak.chr }
```

Zde se vypisují jednotlivé znaky, byte po bytu a rovnou jsou převáděny z ASCII rezprezentace zpět do znakové podoby.

V cyklech někdy potřebujeme použít řídící příkazy. Z Javy nebo Céčka jsou známy příkazy jako **break** a **continue**. Ruby přichází s modifikací a nabízí hned čtyři. **Retry**, **redo**,**next** a **break**. **Retry** spustí v daném okamžiku (splnění podmínky) cyklus znovu od začátku - velmi šikovný příkaz. **Redo** se uplatní při ošetřování libovolné výjimky nebo chyby v průchodu cyklem. Například, pokud by někde přetekla kapacita pole atd. Cyklus ošetříme redem a pokud se splní redo podmínka, cyklus končí - můžeme ho ošetřit i chybovým hlášením. **Next** je alternativou javovského continue. Aniž by prováděl tělo cyklu, přeskočí na další iteraci. **Break** funguje také analogicky. Vyskočí z cyklu.

Cyklus WHILE a DO WHILE, který je v Ruby reprezentován UNTIL fungují zcela intuitivně. Složené závorky prováděcího bloku lze samozřejmě kdykoliv nahradit DO a END. Jen zde popsané možnosti dávají programátorovi značně volnou ruku. I když sám jsem si při programování stejně našel svůj způsob a toho se držel. Zvyk z Javy je železná košile, ulehčuje to trápení a zvyšuje to čitelnost vlastního kódu.

Ošetření výjimek v Ruby

Výjimky jsou standardní nástroj všech vyšších jazyků na odchycení různých nepěkných jevů v aplikacích. Situace, kdy se aplikace dostane do stavu, ze kterého se sama nedostane nebo důsledkem něho úplně "zatuhne". Právě tyto situace musí programátor ošetřit při programování pomocí výjimek. Definuje výjimku, definuje reakci aplikace na tuto výjimku. Většinou se do těla výjimky vkládá celá posloupnost příkazů, která musí být chráněna blokem výjimky. Někdy to může být třeba jen jeden příkaz (např. při otevírání souboru).

Výjimka v Ruby:

```
begin
  ...
rescue Exception
  ...
ensure
  ...
end
```

Je obdobou Javovské konstrukce:

```
try
{
 ...
}
catch (Exception)
...
finaly
...
```

Kapitola 5. Webový framework Ruby on Rails

5.1. Historie

Volně čerpáno z: zdroj [15]

Tento framework pro tvorbu webů vytvořil dánský programátor **David Heinemeier Hansson** pro firmu **37signals**. Narodil se roku 1979 v Copenhagenu v Dánsku. Zde vystudoval univerzitu se zaměřením na obchod. Poté se přestěhoval do Chicaga. Po vytvoření nástroje Ruby on Rails získal titul "Best hacker of the year 2005". V následujícím roce pak získal ocenění "Jolt award of product excellence" za Rails 1.0. Stal se vůdcem komunity kolem Ruby on Rails. On a jeho tým pracuje na nových verzích Railsů, posuzují i přínosy přímo z komunity, které stojí za to implementovat do dalších verzí Rails.V současné době je uvolněn Rails 2.0, který přináší několik změn. Aplikace napsaná ve dřívější verzi není plně kompatibilní s Rails 2.0. Vývojáři byli na změny upozorňováni průběžně od verze 1.5, kdy se některé metody a postupy dostaly na seznam DEPRECATED a ve verzi 2.0 již nefungují. Malým příkladem může být třeba přístup k formulářům, kdy se již nepoužívá dvojice tagů: START_TAG a END_TAG, ale pouze jeden tag vymezující formulář, atd.

5.2. Autor o Ruby on Rails

Rozhovor s Davidem H. Hanssonem pro časopis Software Developer, **citace: zdroj [16]**

Na slovíčko s Davidem H. Hanssonem

Red:

Bruce Tate ve své knize Beyond Java namítá, že platforma Java je na překážku programátorské výkonnosti a nutí programátory, aby se poohlíželi po její alternativě, zřejmě Ruby a Ruby on Rails. Jaký je váš názor na takové hodnocení dnešní platformy Java?

D.H.Hansson:

Bruce Tate trefil do černého. Java urazila za posledních deset let obdivuhodnou cestu. Může být na mnoho pyšná a i nadále tu bude ještě dlouho přítomná. Ale nevěřím, že je pro ni na pomyslném vrcholu ještě místo. Zakořeněná povýšenost a složitosti, s nimiž je třeba se vypořádávat, již začínají unavovat. Nahradíte-li "není možné" variantou "neměli byste, pokud", otevře se vám úžasný svět možností. Svět, v němž se situace a způsoby využití náhle vynořují a přesahují omezenou sadu možných použití, kterou byl v devadesátých letech domněle vševědoucí tvůrce jazyka schopen předpovědět.

Red:

Co zabrání rozhraní Ruby on Rails, aby se vyvíjelo stejným směrem jako Java/J2EE?

D.H.Hansson:

Jeho podstata. Na rozdíl od komerčních subjektů nejsme vydáni na milost či nemilost další velké zakázce. Nesnažíme se vytvořit rozhraní, které by bylo schopné vyřešit oněch pět procent nejobtížnějších problémů světa. V tom spočívá naše silná stránka: ve snaze vyhovět valné většině aplikací, kde lze v široké míře uplatnit společnou sadu konvencí a kde v převážné části své práce většina lidí potřebuje a přeje si totéž. Jsme si proto velmi vědomi skutečnosti, že Rails nikdy u nikoho nevyřeší sto procent problémů. Pokud se nám podaří zeefektivnit alespoň takových osmdesát procent, budete mít více než dost času na vytvoření opravdu skvělé aplikace tím, že budete jedineční v těch zbývajích dvaceti procentech.

Red:

Neexistuje nebezpečí, že uživatelé budou Ruby on Rails aplikovat na úkoly, pro něž není toto rozhraní vhodné, a rozšíří tak platformu neužitečným, rušivým směrem?

D.H.Hansson:

Máte naprostou volnost vést Ruby on Rails pro svůj projekt, jakýmkoliv směrem chcete. Je to otevřený a bezplatný software. Jen neočekávejte, že váš příspěvek začleníme do jádra automaticky. Neustále odmítáme jakékoliv záplaty a nápady týkající se jádra Rails. Proto se tohoto rušivého vlivu příliš neobávám.

5.3. Charakteristika

Ruby on Rails se striktně opírá o jazyk Ruby. Autor D.H. Hansson sám uvádí, že jazyk Ruby se přímo hodí pro toto rozhraní a byla by chyba snažit se vnutit frameworku jiný programovací jazyk. Rails je silně zaměřen na přístup k databázi, na přístup ke konkrétním objektům na webu. Využívá strukturu MVC - Model, View, Controller, o které se dopodrobna zmíním při návrhu a realizaci vlastní aplikace a to přímo na konkrétním příkladu. Přístup do databáze i přístup k formulářům se provádí pomocí modelových tříd. Tyto modelové třídy dokáže Rails vytvořit pro obecné použití zcela automaticky za dodržení daných konvencí.

5.4. MVC obecně

Model, pohled, řadič. To je schéma, které striktně využívá Ruby on Rails. Při vývoji aplikací se musíme držet tohoto schématu, nemáme v podstatě jinou možnost. Na toto základní schéma je nabalena celá funkčnost rozhraní. Všechno má své místo, tak aby Ruby on Rails vědělo, kde načíst jakou část aplikace a jak s ní zacházet.

MODEL

Tato část zodpovídá za výpočetní funkce aplikace a za mapování načítaných struktur na objekty. V modelu najdeme pouze Ruby třídy (soubory *.rb). Jedná se buď o třídy funkcí naší aplikace (nějakého modelu reálné předlohy), nebo o třídy odvozené od **ActionRecord** (mapovací třída pro databázi mySQL). Můžeme zde najít i modelové třídy pro navázání formulářů. Modelové třídy v konkrétní aplikaci Ruby on Rails najdeme v adresáři **app/models**. Pokud umístíme všechny potřebné modelové třídy do tohoto adresáře, nemusíme ani používat obligátní příkaz

require pro "slepení" kódu tříd. Ruby on Rails načte kompletně obsah adresáře app/models a používá jeho třídy najednou. A to jak v přístupu mezi modelovými třídami, tak i v přístupu třeba z řadiče. Require není potřeba, RoR ví přesně, že toto jsou modelové třídy a tak s nimi zachází.

POHLED

Část, která přímo komunikuje s uživatelem (tedy s webovým prohlížečem). V pohledu najdeme pouze soubory *.rhtml, které kombinují HTML a jazyk Ruby. V kódu HTML používáme skriptovací bloky jazyka Ruby, které nám volají modelové funkce nebo přístupy do databáze. V pohledu se většinou jen posílají žádosti (reakce uživatele) přes řadič do výpočetního modelu a nebo se přijímají znovu přes řadič výsledky (odpověď). Tyto se pak zobrazují v pohledu uživateli na webové stránce.

Pro skriptovací bloky platí následující pravidla:

```
<%
```

jakýkoliv kód jazyka Ruby

```
%>
```

```
<%=
```

Zde se používá přímo načtení hodnoty buď z nějaké proměnné, nebo z funkce. Nebo tímto způsobem můžeme vkládat do HTML kódu zkratky, které nám nahradí ve výsledném kódu konkrétní HTML tagy (například start_form_tag) viz formuláře.

```
%>
```

Všechny soubory pohledů najdeme v adresáři **app/views**. V tomto adresáři najdeme podadresář, který se jmenuje stejně jako řadič, pro který chceme pohledy používat. Aplikace může mít i více řadičů, proto oddělení pohledů do konkrétních podadresářů. Že pohledy přísluší řadiči není náhoda. Jednotlivé soubory pohledu jsou přímo svázány s konkrétními akcemi v řadiči. Platí konvence, že pohled se musí jmenovat stejně jako akce v řadiči. Při provádění této akce se pak volá příslušný pohled.

V adresáři **app/views** najdeme ještě jeden podadresář se jménem **layouts**. Tento adresář slouží k uchovávání layoutu pro všechny pohledy konkrétního řadiče. Stačí nazvat layout stejně jako řadič, pro který má platit. Tento layout funguje jako šablona webové stránky, ve které můžeme definovat shodné prvky ze všech stránek. Třeba menu, záhlaví, zápatí, atd. Místo, kam chceme vkládat dynamický obsah (tedy ostatní pohledy) označíme:

```
<%= @content_for_layout %>
```

Podrobnosti uvedu přímo na mé konkrétní aplikaci v kapitole Vývoj konkrétní aplikace.

ŘADIČ

Je zodpovědný za řízení aplikace. Řadič směruje jednotlivé stránky RHTML pohledů pomocí akcí. V jednotlivých akcích přistupuje do modelu a v závislosti na něm odpovídá prohlížeči zobrazováním konkrétních pohledů. Základní akce index definuje akci, která se spustí při zadání názvu (adresy) řadiče do prohlížeče. Řadiče najdeme v adresáři **app/controllers**. Jméno řadiče vychází z pojmenování programátora + _controller.rb. Při zadávání adresy v prohlížeči se používá jen jméno řadiče, automaticky je zavolán soubor jméno_controller.rb z adresáře app/controllers. Toto je výhoda přesně definovaného rozhraní Ruby on Rails i s adresářovou strukturou aplikace.

5.5. Spojení Ruby on Rails s databází (MySQL)

Asi největší úsporu času a kódu nabízí Ruby on Rails při přístupu k databázím. Podle mého názoru Ruby hodně vyniká mezi ostatními jazyky, co se týče přístupu k databázi. Názorně zde vysvětlím a ukáži přístup k databázi MySQL - dnes asi nejpoužívanější databáze jednoduchých i středně složitých webů. Pokud máme nainstalovaný server MySQL, nic nám nebrání začít v práci.

Filozofie přístupu k datům

V nejoblíbenějším jazyce posledních let PHP a podobných jazycích se přistupuje k databázi pomocí databázových dotazů jazyka SQL. Tomuto svrchovanému a jednotnému přístupu lze vytknout jedinou věc. Zbytečně zatěžují programátora, Ruby se nás snaží odstínit od problémů jazyka SQL. Samozřejmě, že nakonec se vše přeloží do výsledné podoby SQL dotazu a ten se provede. To už jde ale mimo režii programátora, tento "překlad" už obstará samo Ruby. K datům z databáze totiž můžeme přistupovat přes objekty. Záznam z tabulky je pro Ruby objektem a tak s ním může programátor zacházet. Při čtení dat nebo i při zápisu přistupujeme k jednotlivým položkám objektu, vůbec nás nemusí zajímat struktura tabulek nebo konkrétní řádky. Samozřejmě se za vším skrývá mapování tabulek na objekty, přičemž musíme při návrhu databáze i při přístupu k ní dodržovat jistá pravidla.

ActiveRecord

ActiveRecord je modul obsahující podporu práce s databázemi v Rails. Najdeme zde třídu **Base**, kterou využívají modely s podporou dat. ActiveRecord::Base zajišťuje připojení k databázi.

Nastavení přístupu k databázi

Pokud chceme přistupovat k databázi, nejdříve musíme naší aplikaci sdělit, do které databáze a pod jakým oprávněním chceme přistupovat. Po vygenerování kostry aplikace Ruby on Rails nalezneme v adresáři aplikace podadresář *config*, ve kterém se nachází soubor *database.yml*. Právě v tomto souboru se nastavuje název databáze, přístupové jméno a heslo. Zajímavostí je, že Ruby používá pro konfiguraci jazyk YAML a ne jazyk XML, který je standardem u ostatních jazyků (pod .NET).

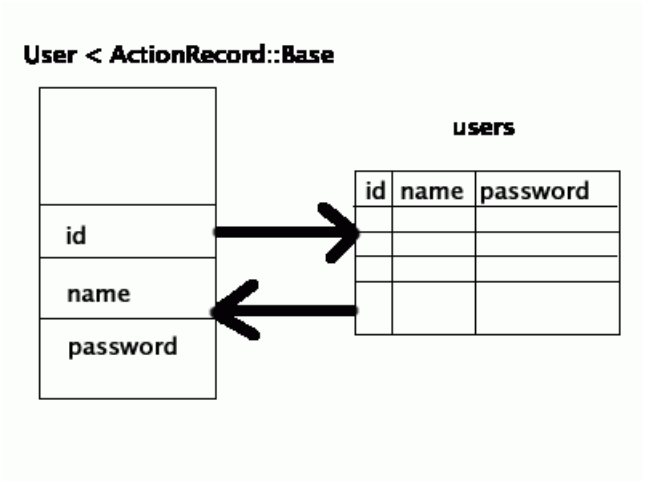
Obrázek 5.1. database.yml

Zde můžeme nastavit přístupy k databázi a to pro testování, pro vývoj i pro konečnou produkci.

Mapování tabulky na objekt

Jak tedy funguje namapování záznamu z tabulky na konkrétní objekt? Pokud chceme takto přistupovat k databázi, musíme konkrétní tabulku pojmenovat stejně jako se jmenuje objekt, na který ji chceme navázat. Přesněji, tabulka musí nést název množného čísla objektu (třídy). Tedy pokud chceme do tabulky **users** přistupovat pomocí objektu user (třída **User**), musíme dodržet oba názvy: Tabulka users a třída User. Zde Ruby používá jakýsi slovník množných podstatných jmen, doporučuje se tedy volit anglické názvy a to především obecná a standardní slova, aby Ruby provedlo správné namapování.

Obrázek 5.2. Mapování tabulky na objekt



Třída User bude dědit právě od třídy Base z modulu ActionRecord, plně postačí:

```
class User < ActiveRecord::Base
end
```

Čtení z databáze:

Stačí použít třídu User a speciální metodu *find(index)*, která vrátí konkrétní objekt, odpovídající řádku s daným indexem v tabulce users.

```
user=User.find(1)
```

V proměnné user bude načtený objekt (pokud se v databázi nachází na indexu 1 nějaký záznam), který bude mít atributy: id, name a password. Tyto můžeme dál libovolně používat.

Zápis do databáze:

Znovu použijeme třídu User a tentokrát metodu *save* pro zápis do databáze.

```
user=User.find(1)
user.name="Pepa"
user.save
```

Do proměnné user se načte objekt - záznam řádku s indexem 1 z tabulky users. Do položky objektu **name** uložíme hodnotu "Pepa". Použitím metody **save** se změna uloží do tabulky v databázi.

Ulehčení - nástroj scaffold

Pro usnadnění tohoto postupu ve složitějších aplikacích existuje v Ruby nástroj **scaffold**. Jedná se o jakési lešení, které vytvoří tento rámec přístupu za vás. Stačí zadat název modelu a řadiče a nechat vygenerovat kostru pomocí scaffold. Z názvu modelu vznikne třída s názvem modelu, která dědí od ActionRecord::Base. Dále scaffold vytvoří napojení na tabulku množného čísla názvu modelu. Do řadiče pak připraví metody pro obsluhu této databázové tabulky, tedy metody pro vkládání, editaci a mazání záznamů. Výstup ze scaffoldu můžeme a nemusíme využít. Při tvorbě jednoduchých vazeb na databázi a při jednoduché obsluze zcela postačí takto automaticky vygenerovaná kostra.

Scaffold v názorné ukázce:

Chceme vygenerovat model a kontroler k předchozímu příkladu. Tedy předpokládejme, že vytvoříme v databázi tabulku **users**, kam se budou ukládat jednotliví uživatelé, k této tabulce bude přistupovat model - tedy třída **User** přes kontroler (řadič) **Sprava**. Nejdříve musíme mít vytvořený rámec aplikace rails, vytvořenou databázi, tabulky, nastavený přístup viz výše a pak se můžeme pustit do generování.

Nejdříve vygenerujeme rámec rails pro naší ukázkovou aplikaci, která se bude jmenovat scaffold:

```
rails scaffold
```

Generátor nám vytvoří základní kostru webové aplikace.

Teď potřebujeme vytvořit databázi s tabulkou. Použijeme tabulku users v databázi sudoku z minulého příkladu. Máme tedy vytvořenou tabulku i nastaven přístup k ní (upravením database.yml).

Pro generování pomocí *scaffold* stačí zadat:

```
ruby script/generate scaffold User Sprava
```

Scaffold nám vytvoří kontroler (řadič) sprava_controller, který obsluhuje jednotlivé akce. Základní akce už jsou v něm nadefinované a to právě ty pro práci s tabulkou users v databázi. Akce pro procházení jednotlivých záznamů, pro vkládání, editaci a pro mazání. Vygenerovaly se i pohledy (views), tedy soubory *.rhtml a to pro všechny akce v řadiči sprava_controller. Současně nám tento nástroj vygeneroval i model user, tedy třídu User, zatím prázdnou, vypadá úplně stejně jako kdybychom jí psali ručně:

```
class User < ActiveRecord::Base
end
```

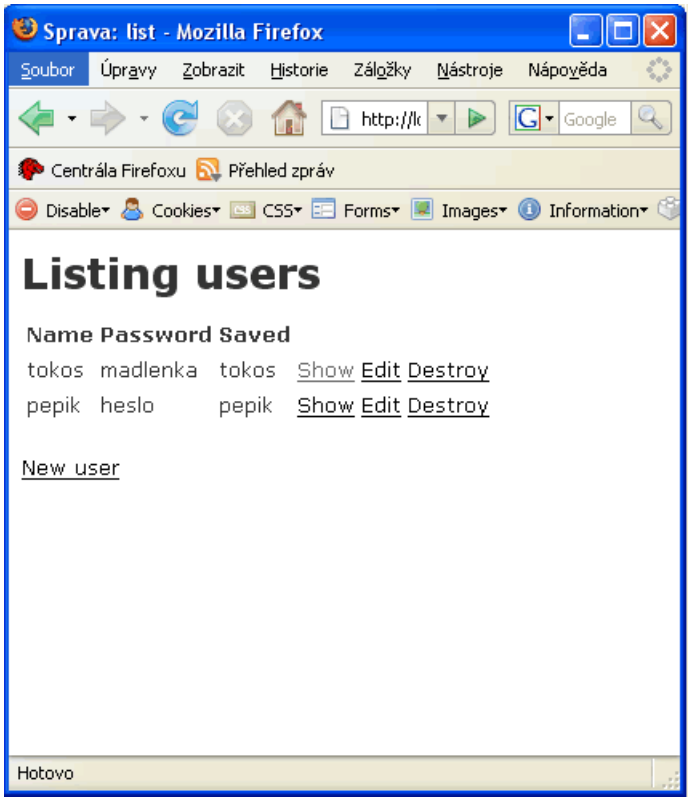
Nyní můžeme aplikaci spustit přes webové rozhraní. Nejdříve spustíme WEB-rick server:

```
ruby script/server
```

A poté zadáme v prohlížeči:

```
http://localhost:3000/sprava
```

Obrázek 5.3. Správa položek



Zobrazí se nám jednoduché webové rozhraní, tedy aplikace zareaguje řadičem sprava_controller, v něm se přerendruje na akci *list* a zobrazí se výsledek akce list (list.rhtml), vidíme i odkazy na další akce: *show*,*destroy* a *edit*.

Při volání akce **list** se řadič zobrazuje pohled **list.rhtml**:

```
<h1>Listing users</h1>

<table>
  <tr>
  <% for column in User.content_columns %>
    <th><%= column.human_name %></th>
  <% end %>
  </tr>

<% for user in @users %>
  <tr>
  <% for column in User.content_columns %>
    <td><%=h user.send(column.name) %></td>
  <% end %>
    <td>
    <%= link_to 'Show', :action => 'show', :id => user %>
    </td>
    <td>
    <%= link_to 'Edit', :action => 'edit', :id => user %>
    </td>
    <td>
    <%= link_to 'Destroy', { :action => 'destroy',
                             :id => user },
    :confirm => 'Are you sure?', :method => :post %>
    </td>
  </tr>
<% end %>
</table>

<%= link_to 'Previous page', {
    :page => @user_pages.current.previous }
  if @user_pages.current.previous %>
<%= link_to 'Next page', {
    :page => @user_pages.current.next }
  if @user_pages.current.next %>

<br />

<%= link_to 'New user', :action => 'new' %>
```

Tento pohled používá přímo třídu User z modelu, přes kterou načítá záznamy z databáze. Současně nabízí odkazy na další akce (tedy řadič na ně bude odpovídat dalšímy pohledy *.rhtml).

Tyto pohledy jsou kombinací jazyka HTML a vlastního Ruby, kostru tvoří HTML a data se dosazují pomocí skriptu (jednoduché cykly for pro průchod kolekcí sloupců v tabulce - objektu). Zajímavou funkci nabízí jazyk Ruby pro stránkování při výpisu položek z databáze. Dole v kódu nalezneme dva linky, které odkazují na předchozí a následující stránku. Do proměnné **@user_pages** se načtou stránky z databáze po kolika prvcích si nadefinujeme. Toto načtení a rozstránkování se provádí pomocí metody **paginate**, kterou přiblížím dál přímo při popisu mé konkrétní aplikace, kde je také použito stránkování.

Kapitola 6. Vývoj aplikací v jazyce Ruby

6.1. Instalace Ruby (Ruby on Rails)

Windows:

Máme několik možností, jak nainstalovat a zprovoznit Ruby. První možnost je použití tzv. Instant Rails nástroje, který za vás připraví vše potřebné. Touto cestou jsem se na začátku vydal já a přijde mi jako nejsnazší způsob. Stačí stáhnout jeden instalační balíček. Projekt Instant Rails vychází z projektu RubyForge. Tento balíček vám do počítače nainstaluje vlastní Ruby, Rails, server Apache a MySQL. S těmito nástroji zvládnete všechno. Po spuštění Instant Rails se okamžitě spustí server Ruby, Apache a MySQL server a můžete začít spouštět kód Ruby přes terminálové okno. Pokud chcete spouštět aplikace Rails, tedy webové aplikace, je nutno spustit ještě server WEB-rick a to pomocí příkazu:

```
ruby script/server
```

Instant Rails download (http://rubyforge.org/projects/instantrails/)

Druhou možnost popisuje Steven Holzner ve své knize Začínáme programovat v Ruby on Rails. Tato instalace také odkazuje na RubyForge. Ale pokud použijete tuto variantu, budete instalovat všechny součásti postupně a zvlášť. Tedy v první řadě stáhnete samotné Ruby, pomocí Ruby instalátoru. (http://rubyinstaller.rubyforge.org) [http://rubyinstaller.rubyforge.org]

Pomocí příkazového řádku pak zprovozníte i Rails, stačí napsat:

```
gem install rails --include dependecies
```

Samozřejmě zbývá ještě nainstalovat server MySQL pro podporu databází. Steven Holzner doporučuje stažení z http://dev.mysql.com. Použití nástroje Instant Rails je tedy poměrně jednodušší a mnohem rychlejší. Pokud chcete mít ale přehled nad instalací, zvolte druhou možnost.

Linux, Unix:

Na Unixových strojích je Ruby standardně ve výbavě. Přistupovat můžete přímo z příkazového řádku příkazem ruby. Najdeme ho standardně ve většině linuxových distribucí.

Mac OS X:

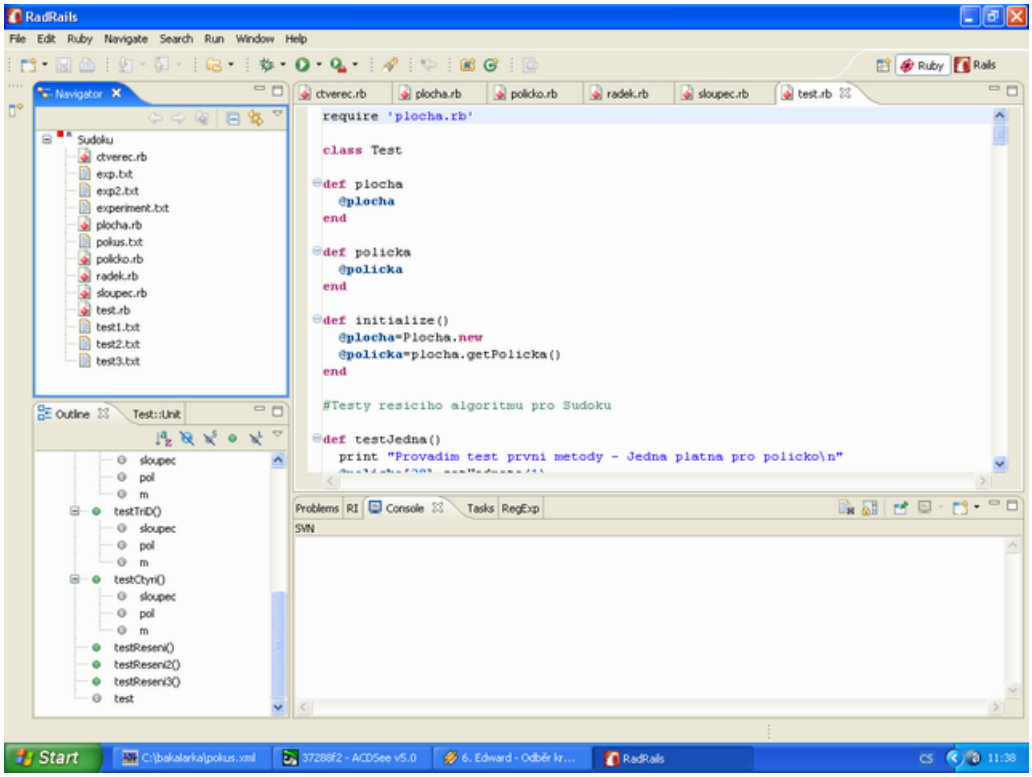
Na Macintoshi najdeme také vestavěnou podporu Ruby podobně jako v Unixu. Samozřejmě můžeme doinstalovat a to znovu z RubyForge, kde najdeme instalaci i pro Mac OS X. Steven Holzner se zmiňuje o malé chybičce na kráse v Mac OS X. Nainstalované Ruby si nerozumí s knihovnou pro readline. Doporučuje se tedy projít vlastní instalací a vestavěnou podporu přeinstalovat.

6.2. Vývojová prostředí - RadRails

Ruby můžeme vyvíjet téměř v jakémkoliv editoru nebo prostředí, ale najdeme různá prostředí, která jsou buď určena převážně nebo výhradně pro Ruby, Ruby on Rails. V dnešní době se Ruby začíná pozvolna dostávat do všech oblíbených a nejpoužívanějších prostředí. Většinou do těch, která se používají pro vývoj v Javě. Dozvíte se tedy o podpoře v Eclipsu a v NetBeans. Osobně bych nejvíc uvítal podporu Ruby v samotném Visual Studiu od Microsoftu, které považuji za nejpovedenější vývojový nástroj. Bohužel, zde se zatím podpora Ruby nevede.

Já jsem si vybral pro vývoj své aplikace prostředí **RadRails**. Nenáročné a poměrně jednoduché prostředí, které ale zvládne "skoro" všechno to, co dokáže i Eclipse. Jediné, co mi opravdu v RadRails chybí je krokování kódu. Najdete zde sice jakousi volbu trace. Pokud se rozhodnete spustit aplikaci v krokovacím módu, spustí se listener na určitém portu, aplikace běží, ale nenašel jsem způsob, jak RadRails donutit krokovat. Tato složitost editor degraduje okamžitě o několik úrovní níže, protože vás odkáže na používání ladících výpisů a jiných metod skoupých na přehlednost.

Obrázek 6.1. RadRails



Vlevo najdeme okno Navigátoru, který zobrazuje součásti otevřeného projektu. Nabídka Outline nám ukáže jednotlivé proměnné a metody daného objektu (otevřeného souboru *.rb v hlavním okně). Ve spodní části nalezneme i plnohodnotnou konzoli. Pokud tedy správně nastavíme prostředí RadRails, můžeme používat i přímo tuto konzoli pro přístup k serveru Ruby.

Instalace RadRails

Balíček ke stažení editoru najdeme na SourceForge.NET, který tento projekt oficiálně podporuje. Průběžně přibývají nové verze s opravami RadRails.

http://sourceforge.net/projects/radrails/

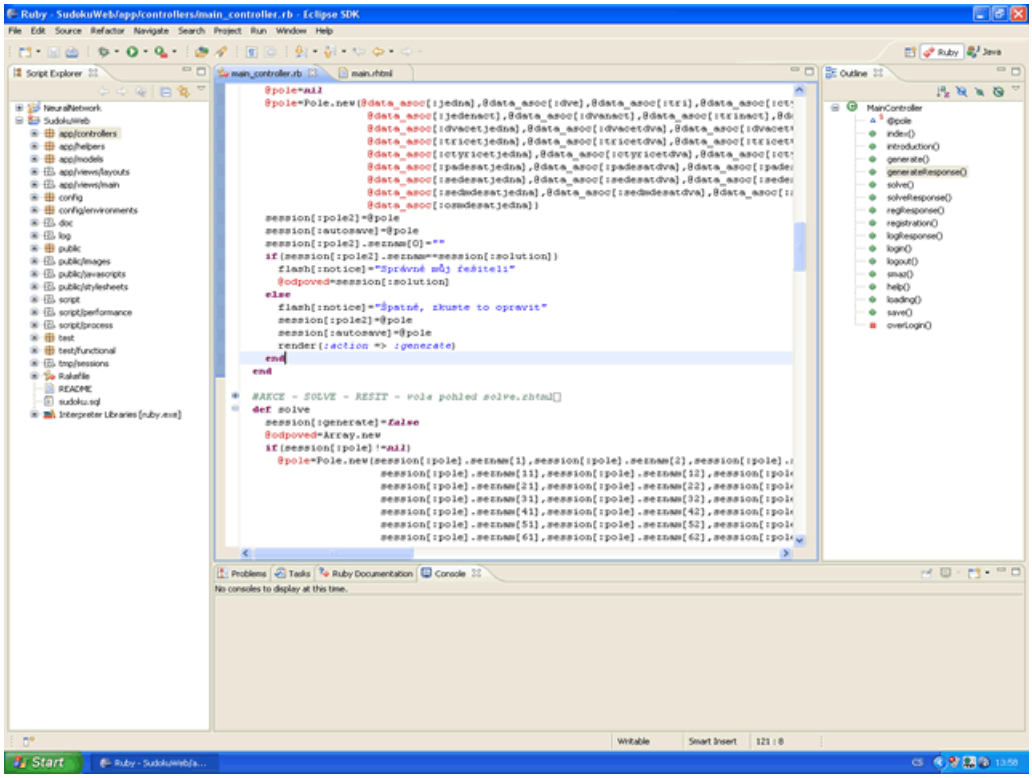
Ve Windows provedeme po stažení jednoduchou instalaci krok za krokem a máme k dispozici přijatelné vývojové prostředí k práci v Ruby on Rails.

Možnosti a funkce

6.3. Vývojová prostředí - Eclipse

Prostředí Eclipse patří ke špičce v editorech pro vývoj aplikací. Našel široké uplatnění mezi profesionálními programátory. Především se používá pro vývoj aplikací v Javě. Je to robustní nástroj, který umí vše, co profesionální prohramátor potřebuje. Jeho největší předností je modularita. Pokud máte Eclipse a chcete, aby se choval trochu jinak, stačí stáhnout konkrétní PLUGIN. Pluginů ke stažení je opravdu mnoho. A právě mezi pluginy pro Eclipse najdeme i jeden pro vývoj aplikací v Ruby (potažmo v Ruby on Rails). Podíváme se tedy blíže, co nabízí toto rozšíření nástroje Eclise programátorovi v Ruby.

Obrázek 6.2. Eclipse s pluginem Ruby



Eclipse jako takový již patří mezi vynikající vývojové nástroje, jehož možnosti se přidáním Ruby pluginu rozšíří i o tento jazyk. Programátora především potěší doplňovací nápověda na jakou je zvyklý z prostředí Visual Studia a kterou nabízí i Eclipse. Stačí stisknout klasickou zkratku CTRL+SPACE a je nám nabídnuta nápověda, co lze doplnit. Eclipse zde přímo zachází s pluginem Ruby a doplňuje i metody předpřipravených tříd jazyka Ruby. Tato funkce v RadRails

chybí a proto díky tomuto pluginu vítězí Eclipse v pomyslné soutěži prostředí. Co lze Eclipsu vytknout je jeho malá přehlednost pohledových souborů *.rhtml, které nemají žádné své přednastavené barevné schéma. Jinak si v Eclipsu můžeme nastavit všechny debugovací nástroje a spouštět přímo Ruby skripty k testování.

Instalace Eclipse:

Konkrétní balíčky najdeme na stránkách projektu Eclipse

http://www.eclipse.org/downloads/

Pro vývoj v Ruby stačí po nainstalování základního balíku stáhnout příslušné pluginy:

Ruby Development Tools - zahrnuje editor, debugger, Ruby IDE

6.4. Vývojová prostředí - NetBeans - JRuby

JRuby

Asi nejpropracovanější vývojové prostředí pro jazyk Ruby nabízí projekt NetBeans od verze 6.0. Sám jsem měl možnost vyzkoušet aktuální verzi 6.01. Projekt NetBeans je od začátku spojen s firmou Sun a tedy i s Javou. Od toho se samozřejmě odvíjí způsob, kterým byla zakomponována podpora Ruby do IDE NetBeans. Byl vytvořen vlastní interpret jazyka Ruby s názvem JRuby. Tento interpret je provázán s knihovnami Javy. Můžeme používat celou řadu standardních knihoven jako je například i knihovna SWING. Ve verzi JRuby 1.0 přibyla i lepší spolupráce s virtuálním strojem Ruby pro JRuby. Tento virtuální stroj se jmenuje Rubinius a nahrazuje v podstatě Javovskou Virtual Machine (JVM).

Informace volně čerpány z článku Root..cz: **Zdroj [17]**

Samotný interpret JRuby je dostupný stránkách projektu a dá se stáhnout i samostatně:

http://jruby.codehaus.org/

JRuby integruje podporu standardních javovských tříd jako tříd Ruby. V kódu k nim přistupujeme jako ke standardním třídám Ruby, které rozšiřují základní balík. Samozřejmě tyto třídy můžeme používat pouze s rozmyslem a nejsou vhodné pro všechny účely, jedná se pouze o doplnění klasických tříd v Ruby. Podpora Javy je rozdělena na dvě skupiny. Na tzv. *low-level* a *high-level*. *Low-level* je implementován v Javě a poskytuje tenký obal nad třídami (java.lang.reflection.* a java.lang.Class). *High-level* je implementován převážně v Ruby a zastřešuje javovskou implementaci *low-levelu*.

Jak přistupujeme tedy k javovským třídám přímo v kódu jazyka Ruby?

Pomocí require načteme podporu Javy a můžeme přistupovat přímo k javovským třídám.

```
require 'java' # Load the high-level Java support

module Foo
# Classes in java.util will be
 include_package 'java.util'

# available in this module

# Create instance of java.util.Random
 r = Random.new

# Write a random integer to output
 puts r.nextInt

end
```

Od verze Ruby 0.8 můžeme používat přímo připojení konkrétní třídy z balíku.

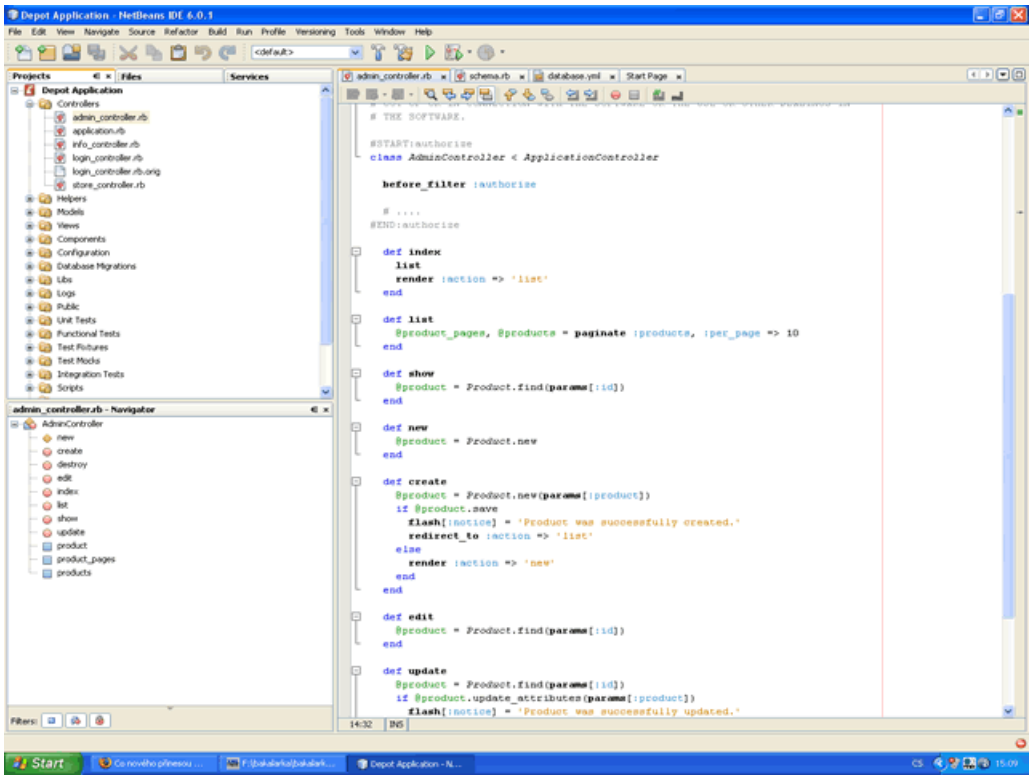
```
require 'java'

include_class 'java.util.Random'

r = Random.new

puts r.nextInt
```

čerpáno přímo ze stránek projektu JRuby. (viz výše)

NetBeans 6.01

Ve vývojovém prostředí tedy už najdeme kompletní podporu pro Ruby a dokonce i pro projekty v Ruby on Rails. Vzhledem k možnostem programátora se jedná o nejpropracovanější prostředí ze všech tří představených. Bohužel NetBeans má jednu nevýhodu. Je velice náročný na hardware počítače a musí mít k dispozici i velký prostor v operační paměti. Pokud ale uživatel nemá problémy s výkonem, je toto IDE opravdu vynikající volbou.

Obrázek 6.3. NetBeans IDE 6.01



V NetBeans nás samozřejmě přivítají všechny možnosti, na které jsme zvyklí z profesionálních prostředí. Intuitivní doplňování kódu, zvýraznění syntaxe Ruby nebo refaktoring, atd. Nabízí i podporu testování a debugu.

Největší zbraní NetBeans je ale vynikající podpora přímo frameworku Ruby on Rails. Můžete založit nový projekt Ruby on Rails. NetBeans vytvoří základní aplikační strukturu a spustí server WEB-rick vaší aplikace. Od této chvíle můžete přes prohlížeč testovat při vývoji vaší aplikaci. Současně máte k dispozici přístup k databázi mySQL. NetBeans spolupracuje i s funkcí **scaffold** (jedná se o skript, generující templaty, řadiče, modelové třídy, atd).

Pro zájemce bych doporučil shlédnout videa přímo od autorů NetBeans ze Sunu, kde je názorně ukázáno, jak pracovat s Ruby on Rails, jak využívat **scaffold**, jak manipulovat s databází, i jak testovat aplikaci přímo v IDE NetBeans.

http://blogs.sun.com/roumen/entry/two_demos_jruby_on_rails

Instalace NetBeans:

Stačí stáhnout kompletní IDE, od verze 6.0 je podpora Ruby a Ruby on Rails zahrnuta přímo v IDE. Žádné pluginy, žádné další balíčky, nainstalujte vývojové prostředí a vyberte si, co chcete vyvíjet.

http://www.netbeans.org/index.html

Kapitola 7. Vývoj konkrétní aplikace: Webový server Sudoku

Kompletní návrh, popis implementace a vlastní realizace projektu webového serveru pro SUDOKU.

7.1. Výběr demonstračního projektu

Rozhodl jsem se vytvořit webový server pro Sudoku jako demonstrační aplikaci vývoje v Ruby. Tato aplikace by měla ukázat většinu možností tohoto jazyka i jeho frameworku pro web Ruby on Rails. Měla by spojit jádro aplikace napsané v jazyce Ruby - vlastní výpočetní model a webovou kostru, vytvořenou pomocí nástroje Rails. Aplikace by měla využívat i MySQL databázi a ukázat tak provázání Rails na MySQL server. Samotný problém Sudoku patří k těm zajímavějším a není tak jednoduchý, jak se může na první pohled zdát. Zvolil jsem tuto aplikaci, abych udělal radost i mému otci, jehož velikou zálibou je právě Sudoku. Právě on se stal mým hlavním testerem aplikace při vývoji. I jeho zkušenosti při každodenním řešení Sudoku mi byly k užitku.

7.2. Požadavky na aplikaci

Aplikace by měla být především schopná řešit SUDOKU podle zadání, které dostane od uživatele přes webové rozhraní. Toto zadání buď zdárně vyřeší, nebo ho prohlásí (s ohledem na své možnosti - výpočetní model) za pro ni neřešitelné v reálném čase. Ve druhém módu by měla aplikace dokázat sama vygenerovat zadání hlavolamu pro uživatele. Tato zadání bude aplikace načítat z databáze a pak k nim přistupovat bez použití výpočetního modelu. Pokud aplikace ve fázi řešení příkladů od uživatele úspěšně vyřeší daný úkol, tento úkol si uloží do databáze jako hotové zadání a později z této databáze načítá náhodnou položku při generování zadání. Uživatele si aplikace rozdělí na přihlášené a nepřihlášené. Přihlášení uživatelé budou mít vytvořen jakýsi účet - položky v databázi, kam se budou ukládat rozpracovaná sudoku. Uživatel bude mít možnost uložit si sudoku při jeho řešení a později se k němu vrátit. Komunikace s uživatelem by měla probíhat přes webové rozhraní - USER FRIENDLY.

7.3. SUDOKU problém

Historie SUDOKU

doslovná citace: **zdroj [8]**

Hra sudoku má předchůdce ve staré čínské hře Lo-šu stejně jako v Eulerových Latinských čtvercích. V 80. letech 20. století začala hra ve své současné podobě vycházet v americkém hádankářském časopise Number Place. Odtud se popularita hry přenesla do Japonska. Vlastní mánie začala ale až poté, kdy si ji zde v roce 1997 povšiml novozélandský turista Wayne Gould, jenž nejen propadl řešení rébusu, ale také vyvinul program, který dokázal hlavolamy generovat. Program bezplatně nabídl londýnským The Times. Na podzim 2004 začaly hádanky vycházet v

Británii a mánie se brzy přenesla i do dalších evropských zemí. V červnu letošního roku vyšla hádanka poprvé i u nás - v Lidových novinách. Na konci léta se pak na českém trhu objevily i první knižní tištěné sborníky.

Klasické SUDOKU:

Sestává z 3x3 čtvreců, každý po 3x3 políčkách, tedy z hracího pole 9x9 políček. Přičemž musí být splněny podmínky:

1. Políčka mohou obsahovat pouze čísla 1-9.

2. V každé řádce se může vyskytovat jedno číslo z tohoto rozsahu právě jednou.

3. V každém sloupci se může vyskytovat jedno číslo z tohoto rozsahu právě jednou.

4. V každém čtverci (myšleno zvýrazněné 3x3 políčka) se může vyskytovat konkrétní číslo právě jednou.

Každý útvar - řádek, sloupec nebo čtverec tedy musí obsahovat všechna čísla od 1 do 9.

Obrázek 7.1. Klasická úloha Sudoku 9x9

9				6				3
1		5		9	3	2		6
	4			5				9
8						4	7	1
		4	8	7				
7		2	6		1			8
2								
5				3	2		9	4
	8	7		1	6	3	5	

Matematika SUDOKU

Sudoku, jak víme, se v základu skládá z hracího pole 9x9 čtverců. Podle pravidel se v něm musí vyskytovat jen čísla od 1 do 9. Na řádku ani ve sloupci ani v bloku 3x3 se nesmí opakovat. Hlavní otázkou zůstává definice problému SUDOKU. Patří mezi NP problémy, nebo už mezi NP úplné problémy, které už nemůžeme deterministicky vyřešit? Problém se mění s rozměrem mřížky. Základní hrací pole má 9x9 čtverců, tedy konečný rozměr n x n. Pokud chceme ale definovat problém pro obecné n x n, pak už se dostáváme do oblasti NP úplných problémů (nedeterministicky polynomiálních úplných problémů). Tím se speciální případy sudoku řadí k NPC problémům jako je problém obchodního cestujícího, hledání nejkratší Hamiltonovské kružnice v grafu, atd. Na řešení těchto problémů se používají různé aproximační algoritmy, genetické algoritmy, nebo Heuristické analýzy.

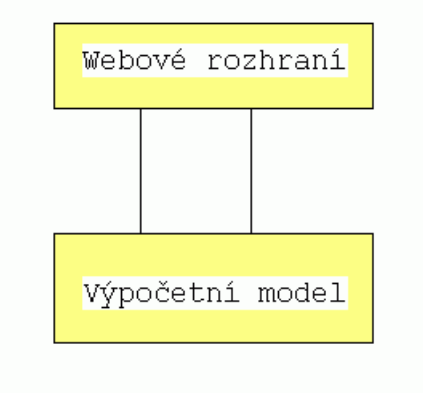
V následujícím odstavci použity informace: **zdroj [6]**

Bertram Felgenhauer (katedra informatiky - Department of Computer Science, Drážďany, Německo) definoval společně s **Frazerem Jarvisem** (katedra matematiky - Sheffiledská univerzita) základní matematiku sudoku. Zaobírali se problémem, jak spočítat počet možných správně vyplněných sudoku mřížek. Sudoku je speciální případ Latinského čtverce. Tento problém se nedá definovat pomocí konkrétní kombinatorické formule. Latinské čtverce do velikosti 11x11 byly spočteny a to pomocí algoritmů BRUTE FORCE, tedy algoritmů, které zkoumají a procházejí všechna možná řešení. Podobně je známo, že mřížka 9x9 Latinských čtverců má 5524751496156892842531225600 = 5.525 × 10^27 možností.

7.4. Návrh aplikace

Bloková schémata a moduly:

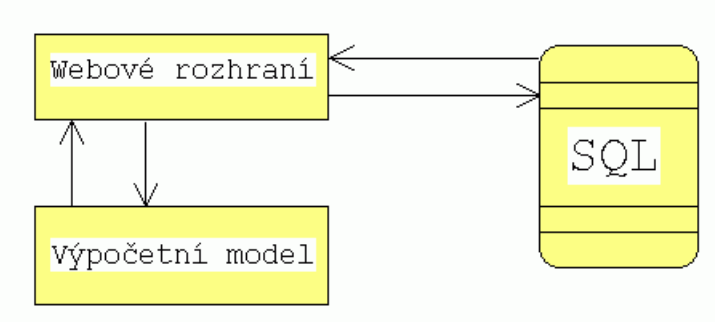
Obrázek 7.2. Základní schéma aplikace



Aplikace v základu staví na dvou oddělených, tedy nezávislých modulech, které si pouze předávají navzájem data. Webové rozhraní komunikuje s uživatelem, přebírá od něj vstupy a ukazuje mu výstupy z výpočetního modelu. Výpočetní model sám o sobě je skupina tříd - samostatná aplikace v jazyce Ruby.

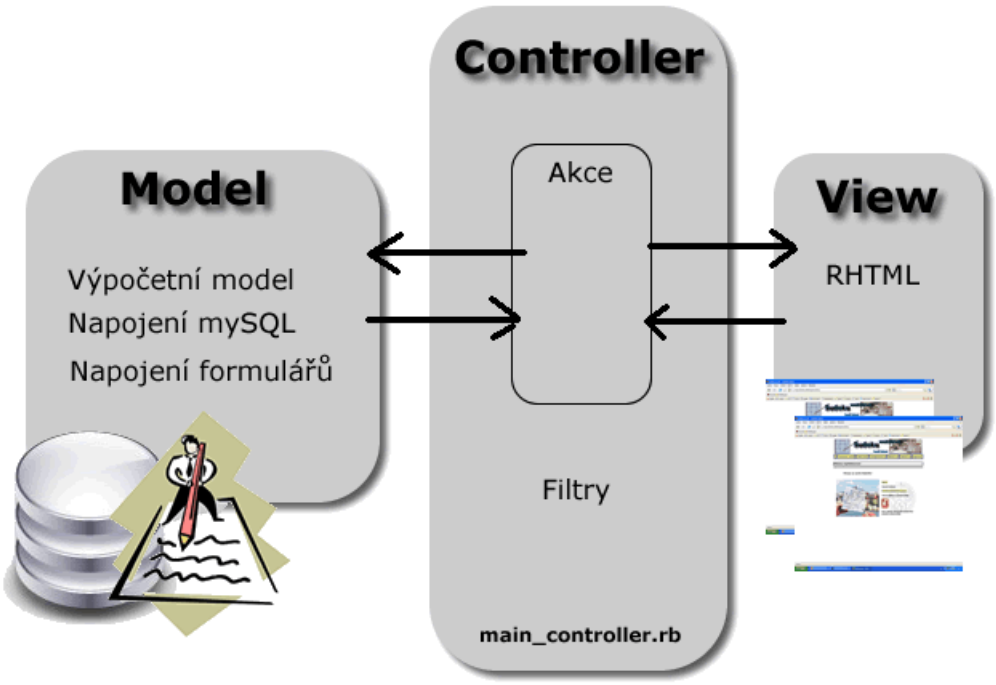
Webové rozhraní pak samostatně komunikuje a využívá ještě MySQL databázi pro přihlašování uživatelů a ukládání výsledků. Kompletní blokové schéma aplikace tedy vypadá takto:

Obrázek 7.3. Rozšířené schéma (Databáze MySQL)

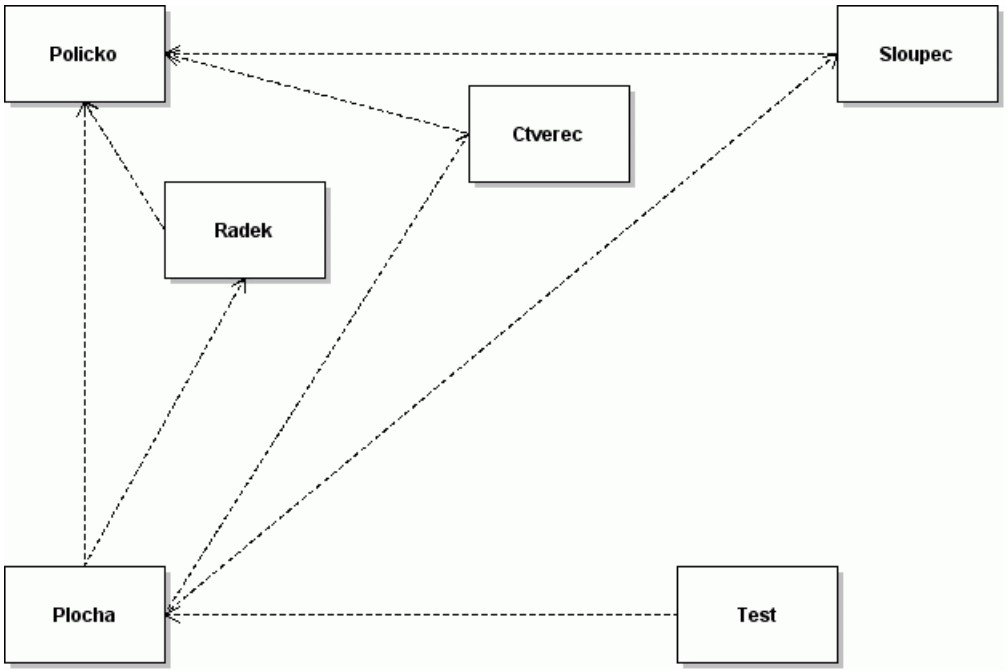


Tak jsem si aplikaci představoval já. Bylo třeba přizpůsobit se ale schématu Ruby on Rails. Aplikaci vyvíjet ve filosofii MVC. Z pohledu Model, view, controller aplikace vypadá takto:

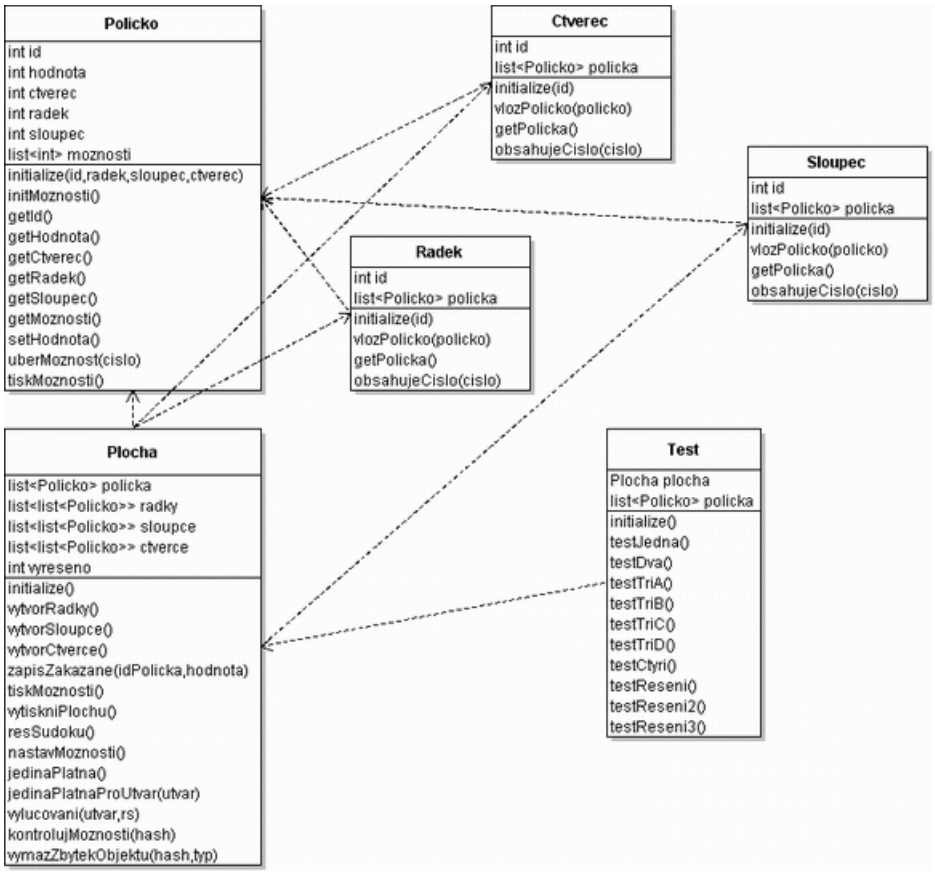
Obrázek 7.4. Struktura aplikace v MVC



Obrázek 7.5. Výpočetní model, schéma tříd



Obrázek 7.6. Rozšířené schéma tříd

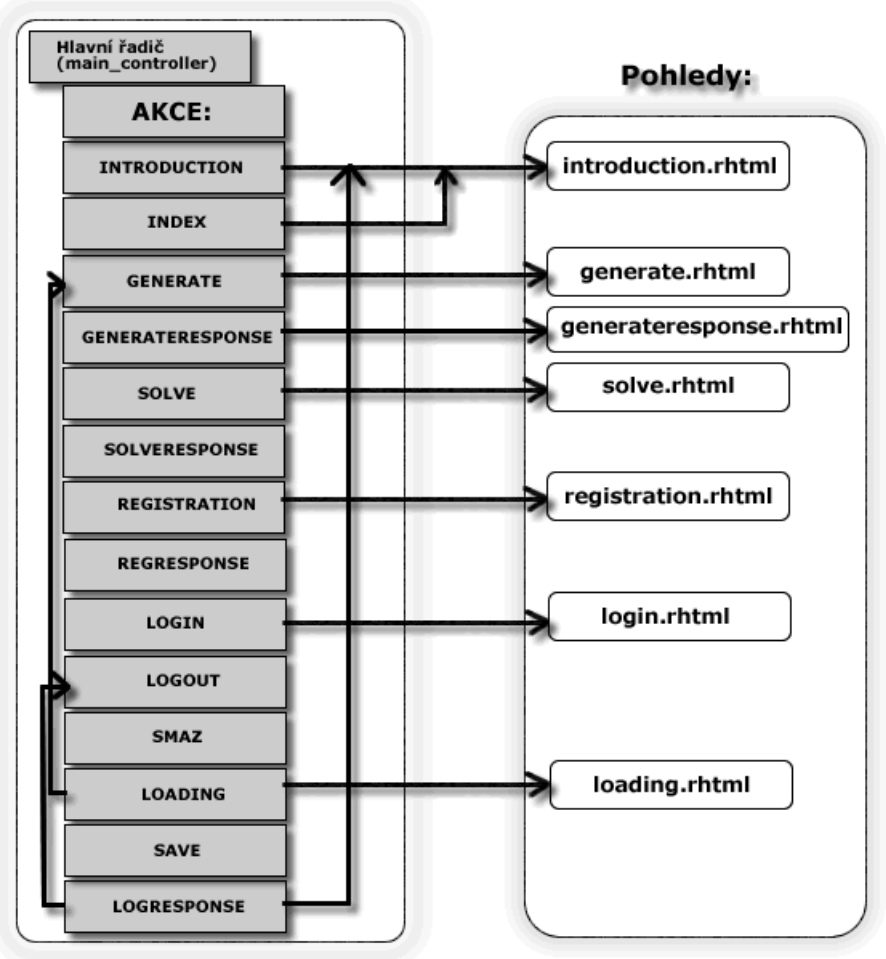


Z tohoto návrhu jsem vycházel při původní implementaci. Ve výsledné aplikaci došlo ke změnám (rozeberu je dopodrobna v kapitole **Metodika a popis tříd**). Změny se týkají především třídy **Plocha**, kde jsem vyřadil jeden výpočetní algoritmus (vylučování) a samozřejmě odebral testovací metody a metody ladicích výpisů. Kompletní zdrojové kódy z původního návrhu zůstaly zachovány a najdete je na přiloženém CD v adresáři **oldcodes**. Adresář **oldcodes/Sudoku** obsahuje úplně první implementaci Sudoku. Jedná se pouze o konzolovou aplikaci, která byla mým prubířským kamenem v cestě dál.

7.5. Návrh webového rozhraní (hlavní řadič)

Webové rozhraní vychází z vlastní filozofie Ruby on Rails. **Staví na struktuře MVC - Model, View, Controller**, tedy model, pohled a řadič. Aplikace obsahuje hlavní řadič (main_controller), který celou aplikaci řídí a obsluhuje. Tento řadič volá konkrétní AKCE na základě vstupů od uživatele nebo výstupů z matematického modelu. Jednotlivé akce pak pro svou komunikaci s uživatelem používají POHLEDY - tedy vlastní internetové stránky reprezentované souborem .rhtml. Webové rozhraní musí umět přistupovat i do databáze (potřebujeme ukládat a číst data, registrovat a validovat přihlašování uživatelů). Za přístup do databáze jsou odpovědné konkrétní AKCE, které k databázi přistupují přes modely (Třídy, které mapují přímo tabulky z databáze).

Obrázek 7.7. Návrh webového rozhraní



Podrobný popis jednotlivých akcí řadiče main_controller:

Hned na začátku řadiče je použit tzv. **BEFORE_FILTER**, který dokáže filtrovat jednotlivé akce řadiče. V tomto konkrétním případě se jedná o akce **LOADING**, **SAVE**, **SMAZ** a **LOGOUT**. Pro provádění těchto akci musí být uživatel přihlášen. Pokud přihlášen není, tyto akce se odfiltrují s hláškou, že uživatel není přihlášen. Velice šikovná věc na podchycení právě neoprávněného přístupu. Uživatel tento filtr neobejde ani přímým zadáním akce do adresy prohlížeče. Jakmile řadič volá tyto jmenované akce, ještě před spuštěním (before_filter) se volá metoda **overLogin()**, která je dostupná přímo v řadiči. Ta ověřuje, zda je uživatel přihlášen a to za pomoci **SESSION**. Pokud není uživatel přihlášen, tato metoda ho okamžitě přesměruje na akci LOGIN, kde se může přihlásit. Přes tento postup (filtr) se uživatel nedostane, leda by dokázal vytvořit objekt třídy Login se svými přihlašovacími údaji a ten vložit do SESSION[:logged] na serveru ještě před voláním těchto chráněných akcí.

Zavedení before_filtru:

```
before_filter :overLogin,
  :only => [:loading,:save,:smaz,:logout]
```

Metoda overLogin v řadiči main_controller:

```
private
def overLogin
  unless session[:logged]!=nil
    redirect_to(:action => "login")
  end
end
```

INDEX

Tato akce zastupuje základní soubor index.rhtml (html). Definuje, co se má stát po zadání hlavního řadiče do adresy webového prohlížeče. Tedy pro tento konkrétní případ, kdy je spuštěn server WEBRICK na portu 3000, stačí zadat adresu `http://localhost:3000/main` a spustí se akce INDEX. Akce INDEX v této aplikaci je naprogramována zcela pragmaticky a jednoduše, přesměruje se na akci INTRODUCTION.

INTRODUCTION

Tato akce zobrazí v pohledu **introduction.rhtml** jakýsi úvod, tedy pár slov o autorovi, o aplikaci, atd.

GENERATE

Akce pro generování zadání. Tuto akci využívají i další akce a přesměrovávají se na ní za jistých okolností. První možností je click na položku GENEROVAT ZADÁNÍ v menu. V tomto případě akce sáhne do databáze připravených úloh (které mohou být jak předpřipravené, tak získané průběžně od uživatelů) a vybere náhodné zadání, které zobrazí uživateli. Ten může řešit a pokud je hotov, stiskem tlačítka ZKONTROLUJ se dostane na akci GENERATERESPONSE, která ho buď pošle zpět při neúspěchu, nebo mu zobrazí správný výsledek a pogratuluje mu. Na akci GENERATE se můžeme dostat ale i volbou NAČÍST uložené sudoku, pokud jsme přihlášení uživatelé a vybereme si z databáze nějakou předchozí svojí rozpracovanou úlohu. Ta se načte právě do akce GENERATE a pohled **generate.rhtml** jí zobrazí.

Poznámka

Při načítání náhodné položky z databáze úloh je použita klasická konstrukce jazyka Ruby pro vybrání náhodné hodnoty z kolekce. Používá se právě tento postup. Načtu všechny položky z databáze do seznamu, ten seřadím **sort_by { rand }** *a vyberu první prvek.*

```
@tasks=Task.find(:all)
@tasks=@tasks.sort_by{ rand }
@task=@tasks.first
```

Toto je ukázka doporučeného způsobu. Samozřejmě existuje více možností, jak z kolekce dostat náhodnou hodnotu, ale Ruby doporučuje právě tento způsob. Použil jsem tedy tuto variantu.

GENERATERESPONSE

Akce odpovídá reakci na odeslání formuláře v akci (spíše navázaném pohledu) GENERATE (**generate.rhtml**). Jedná se o reakci aplikace na stisk tlačítka ZKONTROLUJ, kdy uživatel chce nechat zkontrolovat sudoku, které mu zadal počítač a on ho vyřešil. Akce porovná sudoku s uloženým správným výsledkem. Při neshodě ho přesměruje s hláškou neúspěchu zpět na akci GENERATE a nechá ho dál řešit. Při úspěchu zobrazí i správný výsledek ve vedlejší tabulce.

SOLVE

Výběrem ZADAT ÚLOHU v menu se uživatel dostane na akci SOLVE. Tato akce úzce využívá třídy modelu: Policko, Plocha, Radek, Sloupec, Ctverec a Sudoku. Částečně jsou vstupy z pohledu **solve.rhtml** ošetřeny už při modelovém zpracování formuláře, jeho vstupy se totiž mapují přímo do objektu třídy Pole. Tato třída abstrahuje jednotlivá políčka v herním sudoku, ne však pro výpočetní model, ale pro modul webového rozhraní, protože právě přes tuto třídu získává aplikace data z vyplněné mřížky od uživatele. Ve třídě Pole najdeme i metodu pro částečnou prvotní kontrolu - ošetření vstupních údajů ze syntaktického hlediska, tedy uživatel musí zadávat pouze čísla od 1 do 9. Pokud zadá písmena, aplikace ho na to upozorní.

Poznámka

Převod stringu na integer v Ruby

Zde, jsem narazil na specialitu Ruby v práci se stringy. Pokud uživatel zadá např. "8a", Ruby při klasickém převodu na INTEGER porozumí tomuto stringu jako OSMIČCE! Tedy pustí takovýto omyl uživatele jako překlep. Pokud by ale string začínal písmenem a ne číslicí, k tomuto přiřazení by nedošlo. V tomto případě by Ruby přiřadil do proměnné místo osmičky nulu.

```
cislo="8a"
cislo2="a8"
print "\n8a.to_i => ",cislo.to_i
print "\na8.to_i => ",cislo2.to_i

8a.to_i => 8

a8.to_i => 0
```

SOLVERESPONSE

Reakce na odeslání formuláře v akci SOLVE, tedy vygenerování odpovědi na uživatelem zadanou úlohu. Pokud výpočetní model (modelové třídy) dokázal vyřešit úlohu, uživateli je zobrazen výsledek a současně je zadání úlohy i její řešení uloženo do databáze (pokud tam již není), aby sloužilo jako zdroj úloh při generování zadání. Pokud aplikace nedokáže problém vyřešit, zobrazí uživateli pouze hlášku, že zadanou úlohu nemůže vyřešit a ať zadá něco lehčího.

REGISTRATION

Dostáváme se k práci s uživatelskými účty. Tato akce je zodpovědná za registraci uživatele. Tedy vytvoření uživatelského účtu. Akce k databázi přistupuje znovu přes modelovou třídu (User) pro spojení s MySQL a používá tabulku **USERS** z databáze SUDOKU. Uživatel si vy-

bere své už. jméno a zvolí si heslo. Aplikace kontroluje výskyt už. jména v databázi, případně uživatele varuje, že dané už. jméno již používá někdo jiný.

REGRESPONSE

Reakce na formulář v pohledu registration.rhtml. Po úspěšné registraci dojde k přesměrování na akci LOGIN, kde se může uživatel rovnou přihlásit. Při neúspěšné registraci je uživatel vrácen zpět a znovu vyplňuje vstupní formulář v pohledu **registration.rhtml** v akci REGISTRATION.

LOGIN

Akce pro zalogování uživatele. V pohledu login.rhtml vyplňuje uživatel své uživatelské jméno a heslo.

LOGRESPONSE

Odpověď na vyplnění jména a hesla od uživatele. Pokud akce najde uživatele v databázi, je přihlášen. To znamená, že se vytvoří objekt třídy Login, do kterého se vloží jeho přihlašovací údaje a tento se vloží do **SESSION[:logged]** na serveru, kde tyto údaje přetrvávají i při směrování mezi akcemi. Tedy server pořád ví o přihlášeném uživateli, současně projde i chráněný režim pomocí before_filtru. Pod menu přibydou další dvě tlačítka, která umožňují v chráněném režimu uživateli ukládat rozdělanou práci nebo načítat uložené úlohy z databáze. Po přihlášení je uživatel přesměrován na akci INTRODUCTION, tedy pohled introduction.rhtml. Pokud přihlášení neproběhne, uživatel je o tom informován a přesměrován zpět na akci LOGIN do přihlašovacího formuláře.

LOGOUT

Chráněná akce, uživatel musí být přihlášen, aby se mohla provádět. Zde je toto omezení použito ze spíše formálního a logického hlediska. Odhlašovat se může pouze někdo, kdo je přihlášen. Pokud by akce ale proběhla a uživatel nebyl přihlášen, vcelku nic by se nestalo. Nedošlo by k ohrožení cizích dat. Jedná se pouze o logickou pojistku. Při odhlášení je uživatel přesměrován na akci LOGIN (pohled - login.rhtml) a ze SESSION je odstraněn objekt třídy Login s jeho přihlašovacími údaji.

SMAZ

Další chráněná akce (filtrem). Tato akce vymaže uložený záznam z tabulky **STORES**, na kterou je napojená pmocí modelové třídy Store. Tento záznam si uživatel vybírá v nabídce NAČÍST, kdy se vrací ke svým dříve uloženým rozpracovaným zadáním.

LOADING

Akce chráněná before_filtrem, která načítá uložené rozpracované úlohy z tabulky STORES. Uživateli se zobrazí seznam uložených úloh s konkrétními daty jejich uložení a malým náhledem na konkrétní úlohu. K dispozici jsou volby: Načíst a Smazat. Volba Načíst uživatele přesměruje na akci GENERATE a do pohledu generate.rhtml se načte uložené zadání z databáze, které si uživatel sám vybral. Při volbě Smazat se volá akce SMAZ (výše popsáno).

SAVE

Chráněná akce, funguje pouze po přihlášení uživatele. Uživatel tuto akci může použít pouze pokud se nalézá v pohledu generate.rhtml, tedy akce GENERATE. Uložit svojí práci si může, pokud řeší úlohu, kterou mu zadal počítač. Při ukládání do tabulky STORES pracuje s modelovou třídou Store, která zprostředkovává spojení s databází.

Poznámka

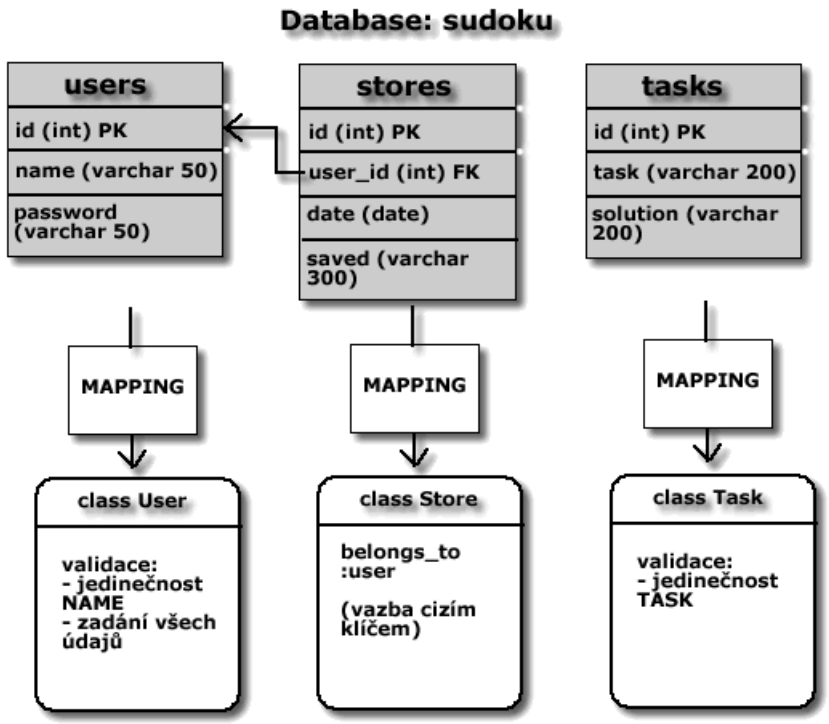
Rezervovaná akce LOAD

Při programování řadiče a jednotlivých akcí jsem narazil na rezervovanou akci s názvem LOAD. Chtěl jsem použít v řadiči akci tohoto jména, ale v praxi to fungovalo tak, že v podstatě nic z této akce po zavolání nefungovalo. Dlouho jsem hledal chybu, až jsem zkusil akci přejmenovat a vše bylo v pořádku. Dopátral jsem se později, že akce LOAD v řadiči je rezervovaná a používá se při spojováním Ruby s technologií Ajax.

7.6. Návrh databáze

Součástí návrhu aplikace je samozřejmě i návrh databáze. Tedy návrh jednotlivých tabulek a principy jejich navázání na aplikaci. Pro ukládání dat mé aplikace postačí plně 3 tabulky MySQL. Pro ukládání přihlašovacích údajů o jednotlivých uživatelích tabulka **USERS**. Pro uchovávání zadání pro uživatele tabulka **TASKS** a odkládací tabulka **STORES**, kam si budou uživatelé odkládat své rozpracované úlohy.

Obrázek 7.8. Návrh databáze



Spojení databáze MySQL s aplikací pomocí modelových tříd:

Tabulka STORES je navázaná na tabulku USERS pomocí tzv. **CIZÍHO KLÍČE - FK (foreign key)**. V jazyce Ruby tuto skutečnost vyjádříme v modelové třídě Store vazbou **belongs_to :user**. Tato definice se píše do příslušné modelové třídy. Tedy do třídy Store.

Třída User:

```
class User < ActiveRecord::Base
  validates_presence_of :name,:password,:message =>
  "Musíte vyplnit všechny položky!"
  validates_uniqueness_of :name,:message =>
  "Toto jméno již používá jiný uživatel, vyberte si jiné."
end
```

Jak již víme, mapování tabulky z databáze provádíme přes modelové třídy. Oddděním od **ActiveRecord::Base**. Tím možnosti Ruby nekončí, můžeme využívat validaci dat před samotným uložením do databáze nebo v jeho průběhu, či po uložení. **Validates_presence_of** se stará o kontrolu parametrů, zda byly zadány všechny. Můžeme i změnit standardní zprávu uživateli. **Validates_uniqueness_of** kontroluje, zda už tento záznam v databázi je nebo ne. Pokud ano, záznam do tabulky neuloží, případně může vrátit uživateli (aplikaci) zprávu. Pokud v databázi tento záznam ještě není, provede se uložení.

Třída Store:

```
class Store < ActiveRecord::Base
  belongs_to :user
end
```

V této modelové třídě provádíme vlastní navázání na tabulku users (spíše na modelovou třídu User) přes cizí klíč user_id. Stačí provázat takto modelové třídy a mít zadefinované primární a cizí klíče v databázi. Ruby už si tabulky naváže samo. Nepotřebujeme tedy přesnou znalost jazyka SQL.

Třída Task:

```
class Task < ActiveRecord::Base
  validates_uniqueness_of :task
end
```

Zde probíhá znovu validace před uložením. Kontroluje se výskyt položky v databázi. Jedná se o akci, kdy uživatel zadá aplikaci úlohu sudoku k řešení a aplikace ji zdárně vyřeší. Tuto vyřešenou úlohu si ukládá do databáze, do tabulky tasks, ale pouze v případě, že záznam ještě v tabulce není. Shoda se kontroluje ve sloupci TASK. De facto je tedy možné, aby v databázi byly dvě různé úlohy sudoku se stejným řešením. Principielně to možné je, protože sudoku může mít v některých případech i více řešení, tato zadání se ale většinou nepoužívají. Aplikace dovoluje i tuto možnost.

7.7. Metodika postupu tvorby aplikace, řešení problémů (popis tříd)

1. Tvorba matematického modelu Sudoku

V první řadě jsem začal programovat modelové třídy reprezentace Sudoku a její výpočetní části. Nejdříve jsem se musel zamyslet nad samotným hlavolamem a posoudit možnosti jeho řešení. Na první pohled se problém jeví jednoduše, ale při bližším zkoumání jsem musel udělat jisté ústupky a kompromisy, abych se vyhnul nejen matematickým problémům, které se Sudoku souvisí. Tedy například více možných řešení "nekorektně" zadané úlohy. U Sudoku existuje jistá podmínka jednoznačnosti úlohy, ale i přesto, že tato podmínka není splněna, nelze s jistotou říci, že úloha bude mít více řešení. Může a nemusí. Nakonec jsem tento problém vypustil, protože jsem fázi generování úlohy serverem vyřešil jako načítání z databáze předpřipravených příkladů.

Při programování modelu jsem vycházel z navrženého schématu. Tedy třídy: **Policko, Radek, Sloupec, Ctverec, Plocha** a speciální třída **Test**, která mi pomáhala testovat jednotlivé metody výpočtu, pro vlastní aplikaci je zcela nepotřebná. Díky ní jsem si ověřoval správnou funkci metod přímo v konzoli nástroje Instant Rails.

Třída Policko: (policko.rb)

Třída **Policko** reprezentuje jedno hrací pole z mřížky Sudoku. Uchovává hodnotu čísla a současně všechny dostupné možnosti zápisu do tohoto políčka. Tedy, pokud máme prázdné políčko, jeho seznam možností bude obsahovat všechny čísla 1..9. Probíhající výpočetní algoritmus se snaží v každém cyklu ubírat tyto možnosti a pokud narazí na políčko, s pouze jedinou možností, tuto možnost tam zapíše. Po zapsání je seznam možností prázdný. Každé políčko nese navíc informace o tom, v jakém se nachází sloupci, v jaké řádce a čtverci. Tyto informace pak používá výpočetní algoritmus.

Třída Radek: (radek.rb)

Třída Radek reprezentuje jednotlivé řádky 0 až 8 v mřížce Sudoku. Obsahuje seznam políček, tedy objektů třídy Policko. Řádek by měl vědět, jaká čísla se v něm již nachází, tedy dokázat zjistit, zda dané číslo obsahuje. Prvotní metoda vlozPolicko() se stará o úvodní inicializaci hracího pole, kdy se vytvoří políčka a odkazy (pointery) na ně se vloží do jednotlivých řádků podle příslušnosti.

Třída Sloupec: (sloupec.rb)

Třída Sloupec analogicky reprezentuje sloupec v mřížce Sudoku. Uchovává seznam políček ve sloupci, obsahuje funkčně stejné metody jako třída Radek.

Třída Ctverec: (ctverec.rb)

Třída Ctverec reprezentuje vždy skupinu 9 políček ve čtverci. Také obsahuje seznam políček a metody pro hledání konkrétní číslice a vkládání políčka.

Třída Plocha: (plocha.rb)

Tato třída využívá předchozí třídy. Dá se říci, že až třída Plocha plně reprezentuje hrací pole Sudoku, protože udržuje informace o všech políčkách v mřížce, o všech řádcích, sloupcích a čtvercích. Zároveň je v ní implementován řešící (tedy doplňovací algoritmus), který pracuje ve 3 fázích. Lépe řečeno ve dvou fázích, 3. fáze zůstala pouze v návrhu, od její implementace jsem později upustil. V první řadě se ve třídě Plocha vytváří samotná hrací plocha. Tedy vytváří se objekty typu Policko, které se vkládají do proměnné policka, dále se vytváří objekty typů Radek, Sloupec a Ctverec. Plocha obsahuje seznam všech řádků, sloupců a čtverců.

Nyní přistoupíme k samotnému dvoufázovému řešení úlohy. **1. fáze** (JEDINAPLATNA) spočívá v tom, že algoritmus prochází všechny políčka hrací plochy a zjišťuje jejich možnosti (kolik čísel je do nich možno zapsat). Pokud narazí na políčko s jedinou možností, tuto možnost zapíše a jedná se o vyřešené pole. Samozřejmě po tomto kroku bychom klasické Sudoku jen stěží vyřešili. Musí tedy nastoupit **2. fáze** (JEDINAPLATNAPROUTVAR). Tady už se do hry zapojí třídy Radek, Sloupec a Ctverec. Algoritmus prochází tyto útvary a hledá prázdná pole, tedy čísla, která v nich ještě nejsou zapsaná. V závisloti na nalezení těchto chybějících čísel upravuje možnosti a po úpravě znovu začíná od 1.fáze. V záloze jsem měl ještě **3.fázi** (VYLUCOVANI). V této fázi by měl algoritmus procházet útvary a hledat tzv. vylučovací závislosti: "Číslo může být pouze tady nebo tady, z čehož vyplývá,.že v rámci pravidel Sudoku už nemůže být jinde!" Toto pravidlo se ukázalo být velmi složité na implementaci pomocí HASH struktur, které jsem si vybral k řešení problému.

Ve výsledku plně postačí 2. fáze řešícího algoritmu. Aplikace je schopná vyřešit skoro všechny úlohy z časopisů a sudoku knížek. Narazil jsem zatím pouze na jeden typ úlohy, kde by se musela použít metoda vylučování. Jedná se o hlavolamy z deníku Právo, v něm najdeme úlohy 3 obtížností. První dvě (lehké a obtížné) vyřeší algoritmus bez problémů, u typu velmi obtížné už nenajde řešení bez metody vylučování. Narazil jsem i na polemiky, že pokud už se musí sudoku řešit metodou vylučování, nejedná se už o klasické správně zadané sudoku. Ale to je pouze relativní názor. Odpovídalo by to situaci, kdy jsem takovýto typ úlohy nenašel ani přímo v knize:

Zdroj [14]

kde jsou také úlohy 3 obtížností, ale můj algoritmus je dokáže vyřešit všechny. Na žádnou tuto úlohu tedy není potřeba použít metoda vylučování. Implementovány a používány jsou tedy pouze metody pro 1. a 2. fázi výpočetního algoritmu, zde jsem musel udělat malý ústupek oproti původnímu návrhu.

Třída Test: (test.rb)

Třídu Test jsem používal pouze pro účely vývoje, tedy testování, ale také je přiložena do výsledného projektu. Modeloval jsem si na ní různé situace, které jsem požadoval řešit po výpočetním algoritmu. Tedy různě vyplněné úseky polí, po průchodu algoritmu jsem zkoumal, zda se správně nastavily možnosti políček a nebo zda algoritmus správně doplnil čísla.

Poznámka

V realizaci aplikace se objevuje ještě třída Sudoku, která realizuje funkci jakéhosi prostředníka mezi výpočetním modelem a webovým rozhraním. Není zahrnuta v původním návrhu modelových tříd, kam ani logicky nepatří. Nepatří ani do webového rozhraní, stojí někde mezi těmito oblastmi a obstarává komunikaci mezi nimi.

Třída Sudoku: (sudoku.rb)

Třída Sudoku byla doprogramována až při existenci webového rozhraní. Slouží jako prostředník mezi webovým rozhraním (řadičem) a výpočetním modelem. Obsahuje metodu pro řešení sudoku, která komunikuje přímo s výpočetním modelem. Pro zjednodušení komunikace byl vybrán tento prostředník, protože samotná třída Plocha by špatně komunikovala přímo s řadičem webového rozhraní. Před vlastním spuštěním řešícího algoritmu je nutné totiž ještě provést kontroly správnosti sudoku od uživatele, které přímo obstarává třída Sudoku, pokud jsou nalezeny "duplicity" - sudoku nevyhovuje pravidlům už v zadání, nevolá se metoda řešSudoku, ale pouze se přes třídu Sudoku upozorní uživatel a ten musí zadání opravit. Dále slouží třída Sudoku ke zjednodušení komunikace na jakýsi standard (nejedná se o žádnou formu XML standardu) ale pouze o přenos požadavku a odpovědi ve formátu pole čísel, které třída Sudoku předpřipraví a inicializuje do třídy Plocha, kde probíhá samotný výpočet.

Tato třída je také používána pro ukládání úloh do databáze, kdy je zadání úlohy v paměti právě ve třídě Sudoku a je správně vyřešeno, je poté uloženo do databáze. V této situaci třída Sudoku pouze používá třídu **Task**, která slouží jako modelová třída pro tabulku **tasks** z databáze.

Metoda ze třídy **Sudoku** pro inicializaci hrací plochy a spuštění řešícího algoritmu:

```
# Metoda pro spuštění řešení SUDOKU
  def res()
    @plocha.nastavMoznosti()
    if(@plocha.resSudoku()==false)
      return false
    else
      @policka=@plocha.getPolicka()
      @policka=@policka.reverse()
      @vysledek=Array.new
      for x in 1..81
        @vysledek[x]=@policka.pop().getHodnota()
      end
    end
  end
```

Pokud algoritmus v reálném čase nenalezne odpověď, což je symbolizováno návratovou hodnotou FALSE, oznámí to uživateli a dál ve výpočtu nepokračuje. Uživatel je požádán o lehčí zadání. Většinu úloh algoritmus dokáže pohodlně vyřešit.

Pokud je Sudoku vyřešeno, uloží se hodnoty políček do pole **@výsledek**, odkud se natahují do pohledu, který vidí uživatel v odpovědi.

Před výpočtem si řadič ještě zažádá třídu **Sudoku** o provedení *validace duplicit*. To znamená, že se hledají duplicity ve sloupci, řádce nebo čtverci:

```
# Metoda na ověření správnosti zadání
#- jedinečnost čísla v řádku, sloupci, čtverci
  def platna
    @radky=@plocha.getRadky()
    @sloupce=@plocha.getSloupce()
    @ctverce=@plocha.getCtverce()
    @stejne=Array.new
    @radky.each {|radek|
      @poles=radek.getPolicka()
      @poles.each {|policko|
        if(policko.getHodnota()!=0)&&
        (@stejne.include?(policko.getHodnota()))
          return false
        else
          @stejne.push(policko.getHodnota())
        end
      }
      @stejne.clear()
    }

    @sloupce.each {|sloupec|
      @poles=sloupec.getPolicka()
      @poles.each {|policko|
        if(policko.getHodnota()!=0)&&
        (@stejne.include?(policko.getHodnota()))
          return false
        else
          @stejne.push(policko.getHodnota())
        end
      }
      @stejne.clear()
    }

    @ctverce.each {|ctverec|
      @poles=ctverec.getPolicka()
      @poles.each {|policko|
        if(policko.getHodnota()!=0)&&
        (@stejne.include?(policko.getHodnota()))
          return false
        else
          @stejne.push(policko.getHodnota())
        end
      }
      @stejne.clear()
    }
```

```
    return true
  end
```

V cyklech se projíždí všechny validované hodnoty v řádcích, sloupcích a čtvercích. Tyto hodnoty se kopírují do pole **@stejne**, ve kterém se při dalším vkládání hodnot hledají duplicity. Stačí, že je nalezena jedna duplicita v prohledávaném útvaru a příkaz

```
return false
```

ukončí tělo cyklu. Všechny možnosti tedy program projde jenom tehdy, pokud v nich nejsou žádné duplicity a je vše v pořádku.

2. Příprava webového rozhraní:

Návrh a popis řadiče webového rozhraní najdete výše v samostatné kapitole, zde bych se rád věnoval problémům, ujasnil některé postupy a důvody, které mě vedly ke konkrétnímu řešení.

Při tvorbě webového rozhraní pro aplikaci jsem se musel naplno oprostit od výpočetní části. Zaměřil jsem se pouze na komunikaci aplikace s uživatelem. Tento modul bude komunikovat s uživatelem, databází i samotným výpočetním modelem. De facto tedy webové rozhraní bude řídit celou aplikaci v prohlížeči a reagovat na požadavky uživatelů. Přemýšlel jsem o reprezentaci samotného hracího pole Sudoku. V jednoduchosti je síla a ujistil jsem se, že pro komunikaci s uživatelem, zadávání úloh i pro generaci zadání mi budou stačit formuláře. Tedy pro hrací pole formulář o 81 polích. Součástí těchto formulářů jsou i ovládací (odesílací) tlačítka.

Poznámka

Při prezentaci mé bakalářské práce jsem zaregistroval dotaz, proč se při vyplňování úlohy (tedy formuláře) nekontrolují políčka průběžně tak, jak je uživatel vyplňuje. Ano, je to jedna z možností, kterou jsem se také zaobíral, ale nakonec jsem se rozhodl pro variantu kdy se kompletně vyplněný formulář odesílá na server pro kontrolu, validaci a vygenerování odpovědi. Samozřejmostí je, že pokud validace neprojde a uživatel je vrácen zpět (musí opravit nějakou hodnotu ve formuláři), formulář se nevymaže, ale zůstane naplněn daty od uživatele. Ten pak může svou chybu pohodlně opravit. Obsah formuláře se udržuje po dobu kontroly na serveru v tzv. ***SESSION****. S tímto řešením se setkáváme na většině webů. Myslím, že kontrola formuláře ještě před odesláním (což je záležitost Javascriptu) nebo kontrola na serveru po jednotlivých formulářových polích je v tomto případě zbytečná. Navíc je tu ještě jeden zásadní důvod proti tomuto postupu. Samotná hra Sudoku. Hráč přece nechce vědět, zda políčko vyplnil dobře nebo špatně, to se má dozvědět až na konci, kdy odešle své řešení a aplikace mu odpoví, zda byl úspěšný nebo ne. Pokud by aplikace "našeptávala" hráči, co kam nesmí vyplňovat, tento efekt by kazil celý princip hry.*

Zpracování formulářů:

Rád bych také objasnil jak se dá v Ruby on Rails zlehka přistupovat k formulářům a ukáži to na mé aplikaci. Obecně se vyplněné formuláře odesílají na server metodami GET/POST. Na server se odesílají data v proměnných, které odpovídají názvům jednotlivých formulářových

polí. V PHP se přistupuje k těmto datům pomocí proměnných GET/POST ["pole_formulare(promenna)"], pomocí asociativních polí. V Ruby on Rails máme možnost kompletně celý formulář nebo jeho jednotlivá pole napojit na objekty. Napojujeme formulář na modelové třídy tak, jako jsme mapovali tabulku přes modelové třídy na objekt. V případě odeslání formuláře na server vznikne asociace formulářových políček (může asociovat jedno nebo i více polí). Pokud touto asociací inicializujeme modelovou třídu, můžeme pohodlně přistupovat k datům a nebo je dokonce jednoduše přednastavit na ještě nevyplněném formuláři.

Jako zástupné tagy pro formulář v pohledu (RHTML) používáme v Ruby on Rails **<%= start_form_tag %>** a **<%= end_form_tag %>** plus další "zkratky", které nahrazují formulářové tagy. Tyto zkratky poté generují v odpovědi serveru klasické formulářové tagy. Programátor jen napíše zkratku a naváže ji na modelovou třídu. Zkratky se většinou dají pohodlně logicky odvodit:

```
<%= text_field ("modelová třída",
       "jméno prvku",{další parametry}) %>
```

zkratka *text_field* nahrazuje klasický tag **<input type="text" />** v HTML. V závorce poté uvádíme název modelové třídy (spíše asociace, ve které budeme moci přistupovat k datům) a samozřejmě jméno prvku, tím ale rozumíme spíše accessor z modelové třídy, do kterého chceme data z formuláře zapsat. V obecné rovině se mluví o metodě z modelové třídy, v praxi se ale jedná přímo o instanční proměnnou. Ve složených závorkách mohou následovat další libovolné parametry jako jsou délka textového políčka, jeho výška, atd.

```
<%= password_field ("modelová třída", "jméno prvku",
{další parametry}) %>
```

zkratka *password_field* nahrazuje klasický HTML tag **<input type="password" />**.

Samozřejmě existují i další zkratky pro *checkboxy*, *listboxy*, atd. Pro ukázku jsem uvedl tyto dvě nejpoužívanější formulářové zkratky.

Nejlepší bude ukázat vše na konkrétním příkladu mé aplikace.

Modelové třídy pro formuláře v Sudokuserveru:

Třída Uzivatel (uzivatel.rb)

```
class Uzivatel
  attr_accessor :id
  attr_accessor :name
  attr_accessor :password
  attr_accessor :passwordCheck

  def initialize(jmeno,heslo,hesloOvereni)
    @name=jmeno
    @password=heslo
    @passwordCheck=hesloOvereni
  end
end
```

Tato třída obsahuje instační proměnné, které jsou dostupné accessorem, tedy jsou přístupné pro zápis i pro čtení. Dále pak vidíme konstruktor, kde se inicializují proměnné předané parametrem.

Samotný formulář v pohledu RHTML vypadá takto (**registration.rhtml**):

```
<%= start_form_tag({:action => "regResponse"},
                   {:method => "post"}) %>
    <table>
    <tr>
    <td class="title">Uživatelské jméno:</td>
    <td><%= text_field ("user","name",{"size" => 20}) %></td>
    </tr>
    <tr>
    <td class="title">Heslo:</td>
    <td><%= password_field
    ("user","password",{"size" => 20,"type" => 'password'}) %>
    </td>
    </tr>
    <tr>
    <td class="title">Potvrzení hesla:</td>
    <td><%= password_field
   ("user","passwordCheck",
     {"size" => 20,"type" => "password"}) %>
    </td>
    </tr>
    <tr>
    <td><input type="submit" value="Zaregistrovat se" />
    </td>
    </tr>
    </table>
    <%= end_form_tag %>
```

Data z formuláře se nám na serveru po odeslání přenesou do asociace pod parametrem s názvem **:user**. Těmito daty z asociace naplníme modelovou třídu Uzivatel a tyto data můžeme dál používat. V tomto konkrétním případě se data zpracovávají souběžně s databází pomocí modelové třídy User a tabulky users. Data si tedy nejprve vyzvedneme v asociaci **params[:user]** a poté jimi plníme modelovou třídu.

```
def regResponse
    @fail=false
    @data_asoc=params[:user]
    if(@data_asoc[:password]!=@data_asoc[:passwordCheck])
      flash[:notice]="Neshoduje se heslo v potvrzení"
      @fail=true
    else
      @user=Uzivatel.new(@data_asoc[:name],@data_asoc[:password],
      @data_asoc[:passwordCheck])
      @us=User.new
      @us.name=@user.name
```

```
      @us.password=@user.password
      if(@us.save)
        flash[:notice]="Registrace proběhla v pořádku.
                        Můžete se přihlásit."
        @fail=false
      else
        flash[:notice]=@us.errors.full_messages.join(', ')
        @fail=true
      end
    end
    if(@fail)
      render(:action => :registration)
    else
      render(:action => :login)
    end
  end
```

Layout řadiče main

Jak jsem se zmínil u popisu architektury **MVC**, Ruby on Rails umožňuje nadefinovat v pohledech základní **layout**, který bude společný pro skupinu pohledů určitého řadiče. V mém konkrétním případě je tímto řadičem main_controller. Řadič main, ke kterému jsou připojeny přes akce pohledy v adresáři **app/views/main**. Do adresáře **app/views/layouts** jsem umístil soubor **main.rhtml**, který je základním layoutem pro řadič main.

```
<html>

<head>
<title>Sudoku server</title>
<%= stylesheet_link_tag 'main' %>
</head>

<body>
    <table align="center" style="width: 700px">
    <tr>
    <td>
    <% if(params[:id]!=nil) %>
    <img src="../../images/titulek.gif" />
    <% else %>
    <img src="../images/titulek.gif" />
    <% end %>
    <table>
    <tr>
    <td class="menu"><%= link_to "|",
    :action => "introduction" %></td>
    <td class="menu"><%= link_to "Generovat zadání",
    :action => "generate" %></td>
    <td class="menu"><%= link_to "Zadat úlohu",
```

```
    :action => "solve" %></td>
    <td class="menu"><%= link_to "Zaregistrovat se",
    :action => "registration" %></td>
    <td class="menu"><%= link_to "Přihlásit se",
    :action => "login" %></td>
    <td class="menu"><%= link_to "Odhlásit se",
    :action => "logout" %></td>
    <td class="menu"><%= link_to "Nápověda",
    :action => "help" %></td>
    </tr>
    </table>
    <div class="mezera"></div>
    <div class="login">Přihlášen: <%= if(session[:logged]==nil)
                            "Nepřihlášený host"
                        else
                            session[:logged].name
                        end %>
    </div>
    <div class="poznamka"><%= flash[:notice] %></div>
        <% if(session[:logged]!=nil) %>
        <table>
        <tr>
            <td class="menu">
            <%= link_to "Načíst Sudoku", :action => "loading" %>
            </td>
            <td class="menu">
            <%= link_to "Uložit stav", :action => "save" %>
            </td>
        </tr>
        </table> <%
           end
         %>
    <%= @content_for_layout %>
    </td>
    </tr>
    </table>
</body>

</html>
```

Najdeme zde společné prvky pro všechny pohledy. Titulkový grafický baner, horizontální menu, které je provázané přímo na akce řadiče. Dále jsou zde informace o přihlášení nebo anonymním přístupu uživatele (tyto jsou načítány ze SESSION). Proměnná flash[:notice] slouží k zobrazování informačních zpráv uživateli. V závislosti na přihlášení uživatele se zobrazí skryté menu pro ukládání a načítání úloh.

Horizontální menu je tvořeno odkazy, které jsou v kódu Ruby zastoupeny zkratkou **link_to**:

```
<%= link_to "|", :action => "introduction" %>
```

V této zkratce je definován v podstatě zástupný HTML tag <a>, jeho parametry: href="introduction" (což je odkaz na akci řadiče).

Prostor pro dynamický obsah uvnitř layoutu je definován:

```
<%= @content_for_layout %>
```

Celý HTML blok je uzavřen </html>. V jednotlivých pohledech už tedy nemusíme zadávat ani společné prvky, ani kostru HTML souboru. Stačí vyplnit dynamický obsah, podle konkrétní akce.

7.8. Popis hotové aplikace

Po spuštění aplikace na běžícím serveru WEB-rick spatříme následující stránku v prohlížeči (port 3000 na localhostu, pokud používáme InstantRails).

```
http://localhost:3000/main
```

Obrázek 7.9. Spuštěná aplikace serveru Sudoku

Kompletní uživatelskou příručku naleznete na CD v příloze bakalářské práce. Ovládání aplikace je intuitivní a z tohoto důvodu umisťuji uživatelskou příručku do přílohy a ne do hlavního textu, ve kterém se spíše věnuji důležitějším věcem a názorným příkladům z jazyka Ruby a frameworku Ruby on Rails.

Aplikace byla testována na hlavolamech z deníku Právo a na souboru hlavolamů z knihy Sudoku na čas (**Zdroj [14]**). Z deníku Právo aplikace dokázala vyřešit dva typy obtížnosti ze tří. Z knihy Sudoku na čas dokázala vyluštit všechny 3 typy zadaných obtížností. Aplikaci mi pomáhali testovat i členové rodiny, kteří zkoušeli luštit zadané Sudoku.

Kapitola 8. Závěr

Rád bych splnil cíle, které jsem si stanovil v úvodu. Doufám, že tato práce poskytla zájemcům o Ruby (Ruby on Rails) pevné základy hlavně pro rozhodnutí, zda zkusit nebo nezkusit Ruby on Rails. Sám jsem si prošel fází, kdy jsem o Ruby a RoR věděl jen velmi málo. Snažil jsem se programovat dynamické weby v jazyku PHP nebo ASP. Po době strávené s Ruby on Rails nemohu říci, že dynamický web už bych dělal jenom v Railsech a v ničem jiném. Každopádně mohu zodpovědně říci, že jazyk PHP dostal ve srovnání s RoR hodně na frak a úplně jsem ho do budoucnosti zavrhl. V Railsech stvoříte aplikaci dílem okamžiku (po realizaci nějakého testovacího projektu). Musíte se naučit jazyk Ruby, který ale vůbec není těžký. Ona výhoda **"dynamicky typovaného"** jazyka se mi (jakožto programátorovi, který vyrostl na Javě) jeví spíše jako nevýhoda. Pokud bych dělal velký profesionální projekt, asi určitě bych se rozhodl třeba pro JSP - JavaServer Pages. Java jako základ je sice striktní a upovídanější než Ruby, ale právě proto je její kód jednoznačný a velmi snadno v něm s trochou praxe hledáme chyby. Pokud se budeme při programování v RoR trochu více hlídat, není problém vytvořit profesionální aplikaci postavenou na jazyku Ruby.

Poznámka

Dynamicky typovaný jazyk ve smyslu zacházení s proměnnými. Jejich typ je určen až hodnotou, kterou je proměnná naplněna a ne implicitní deklarací typu. Tento problém je přiblížen v kapitole pojednávající o charakteristických rysech jazyka Ruby.

Další věcí, kterou je třeba zmínit je podpora Ruby. V současné době probíhá v České Republice boom vývoje v Ruby on Rails. Se značným zpožděním doháníme svět. Blogy o RoR a různé diskuze vyrostly jako houby po dešti, ale to hlavní stále ještě chybí. Pro masivní rozšíření je třeba, aby v tomto frameworku začali programovat mladí vývojáři a programátoři-amatéři, ze kterých vyroste další generace. V ČR není problém najít FREE webhostingy s podporou jazyka PHP a databáze MySQL. Pokud budete ale hledat free webhosting, kde by běžel i server Ruby, budete hledat marně. Najdete pár placených serverů, které již poskytují podporu Ruby a Ruby on Rails, ale pro masivní rozšíření je třeba víc. Stejně tak jako dokumentace k jednotlivým třídám v základu Ruby. Velmi často totiž musíte zabrousit do různých diskuzí a blogů, abyste pochopili použití třídy a jejích metod. Vytvoření dokumentace pro Ruby (srovnatelné třeba s dokumentací pro Javu) by vydalo na další tři bakalářské práce, rozhodně by ale pomohlo rozšířit Ruby mezi širší masu programátorů. Free webhosting v Ruby jsem marně hledal i ve světě. Narozdíl od ČR si ale můžeme vybrat z nepřeberného množství serverů, které podporují Ruby a které nejsou ani zdaleka tak drahé na pronájem jako ty naše. Jeden Free webhosting se mi přece jen podařilo najít, jedná se ale bohužel o jakýsi rumunský server, který je momentálně nedostupný (pokud někdy vůbec dostupný byl).

Jak je na tom Ruby s praktickým využitím u nás? Našel jsem jednu firmu iQuest, která tuto technologii dlouhodobě využívá. Pro aplikace typu klient/server využívají Javu (JSP) a pro ostatní dynamické webové aplikace používají výhradně framework Ruby on Rails, na který přešli od PHP (a to již v první verzi Ruby on Rails, která se dostala do Čech). Jinak vím ještě o plzeňské firmě UNICORN, která v současné době realizuje jeden projekt v Ruby on Rails. Jedná se ale o výjimku, drtivá většina aplikací od této firmy je napsána v JSP.

Firma iQuest: `http://www.iquest.cz/`

Firma Unicorn: `http://www.unicorn.eu/cz/`

Na otázku, zda vyzkoušet Ruby on Rails odpovídám určitě ANO! Získáte velice propracovaný nástroj podle rčení: "Za málo peněz, hodně muziky." Rozhodně se tato zkušenost vyplatí každému, kdo potřebuje vyvíjet webové aplikace. Ruby on Rails určitě není ztrátou času. Pro důkladné pochopení je třeba napsat si svůj vlastní startovací projekt. Přečtení této práce z nikoho přes noc programátora v RoR neudělá. Moje práce neobsáhla tento framework kompletně. Najdete zde vysvětlení obvykle používaných metod a principů. Námětem na prohloubení znalostí o frameworku RoR mohou být např. filtry. Zmínil jsem se na příkladu o before_filtru. Můžeme používat ale i další typy filtrů a jak jsem již naznačil, právě filtry jsou velmi efektivní zbraní. V této práci byste měli najít podklad pro vaše bližší seznámení s Ruby. Mohu vám slíbit, že při vývoji v RoR si troufnete daleko na víc než kdy předtím a při programování v Ruby se budete opravdu cítit pohodlně.

Co se týče prostředí (editorů) pro vývoj v Ruby (RoR), našel jsem jasného vítěze ze tří testovaných. Jedná se o NetBeans od verze 6.0. Vzhledem k tomu, že Ruby je skriptovací jazyk a odpadá potřeba kompilovat průběžně při vývoji a testování kód, můžeme psát kód doslova v čem chceme. Stačí libovolný textový editor. Tato věta se sice používá dost často i u klasických kompilovaných jazyků, ale tam potřebujeme před spuštěním ještě kompilaci. U Ruby nám stačí kód spustit na serveru Ruby. Při vývoji Ruby on Rails stačí zapnout server WEB-rick a můžeme rovnou výsledek kontrolovat v prohlížeči. Pokud chcete ale prostředí, které vám velice usnadní život, zvolte nové NetBeans a nebudete litovat. U Netbeans není podpora Ruby on Rails pouhou formalitou v dokumentaci a přidáním interpretu Ruby do IDE. Nabízí nám všechny finty frameworku Ruby on Rails pěkně pohromadě.

Kapitola 9. Použité zdroje

Zde najdete kompletní seznam zdrojů vždy pod konkrétním inventárním číslem, které je zmíněno v textu práce a odkazuje právě na zdroj z tohoto seznamu.

9.1. Seznam použité literatury

[9] HOLZNER, Steven. Začínáme programovat v Ruby on Rails. Martin Domes; Karel Voráček. 1st edition. Brno : Computer press, a.s., 2007. 384 s., zdrojové kódy na internetu. ISBN 978-80-251-1630-2.

[14] Sudoku na čas : změř si svůj výkon. 1. vyd. Havlíčkův Brod : Fragment, 2006. 80 s. ISBN 80-253-0280-6.

[16] Na slovíčko s D.H. Hanssonem. Software developer : Magazín pro vývojáře aplikací. 1.1.2006, roč. 2006, č. 1, s. 19-19.

9.2. Ostatní zdroje

[1] MINAŘÍK, Karel. Ruby on Rails a revoluce ve vývoji pro web. Část první: Ruby. Weblog : Rails [online]. 2007 [cit. 2008-02-26]. Dostupný z WWW: <http://blog.karmi.cz/2007/6/16/co-je-ruby-on-rails-cast-1>.

[2] DR. BEZROUKOV, Nikolai. A Slightly Skeptical View on Scripting Languages. Softpanorama [online]. 2006 [cit. 2008-02-26]. Dostupný z WWW: <http://www.softpanorama.org/Articles/a_slightly_skeptical_view_on_scripting_languages.shtml>.

[3] OUSTERHOUT , John K. Scripting : Higher Level Programming. EEE Computer magazine [online]. 1998 [cit. 2008-02-26]. Dostupný z WWW: <http://home.pacbell.net/ouster/scripting.html>.

[4] PRECHELT, Lutz. Are Scripting Languages Any Good?. Článek [online]. 2002 [cit. 2008-02-26]. Dostupný z WWW: <http://page.mi.fu-berlin.de/prechelt/Biblio/jccpprt2_advances2003.pdf>.

[5] JUNGWIRTH, Tomáš. Objektově orientovaný jazyk RUBY [online]. 2001 [cit. 2008-02-26]. Text v češtině. Dostupný z WWW: <http://nb.vse.cz/~zelenyj/it380/eseje/xjunt04/OOJ-Ruby.htm>.

[6] FELGENHAUER, Bertram, JARVIS, Frazer. Enumerating possible Sudoku grids. Článek [online]. 2005 [cit. 2008-02-26]. Dostupný z WWW: <http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf>.

[7] VENNERS, Bill. The Philosophy of Ruby : A Conversation with Yukihiro Matsumoto. Aritma developer [online]. 2003 [cit. 2008-02-26]. Dostupný z WWW: <http://www.artima.com/intv/ruby.html>.

[8] HOUSER, Pavel. Sudokománie pod lupou. Science world [online]. 2008 [cit. 2008-02-26]. Dostupný z WWW: <http://www.scienceworld.cz/sw.nsf/ID/6B2256EA7BC78D21C1257093005DE338?OpenDocument&cast=1>.

[10] ŠŤASTNÝ, Jakub. Ruby on Rails 2.0 : Evoluce, nikoliv revoluce. Root.cz : linuxový server [online]. 2007, roč. 2007 [cit. 2007-12-13], s. 1-1. Dostupný z WWW: <http://www.root.cz/clanky/ruby-on-rails-2-0-evoluce-nikoliv-revoluce/>. ISSN 1212-8309.

[11] MOLIČ, Jan. Ruby on Rails : Úvod. Root.cz : linuxový server [online]. 2005, roč. 2005 [cit. 2005-11-11], s. 1-1. Dostupný z WWW: <http://www.root.cz/clanky/ruby-on-rails-uvod/>. ISSN 1212-8309.

[12] ŠŤASTNÝ, Jakub. 3rdRail : konečně pořádné IDE pro Ruby on Rails?. Root.cz : linuxový server [online]. 2007, roč. 2007 [cit. 2007-10-22], s. 1-1. Dostupný z WWW: <http://www.root.cz/clanky/3rdrail-konecne-poradne-ide-pro-ruby-on-rails/>. ISSN 1212-8309.

[13] ŠŤASTNÝ, Jakub. Jan Minárik: S Rails vyvíjíme efektivněji : rozhovor. Root.cz : linuxový server [online]. 2007, roč. 2007 [cit. 2007-07-04], s. 1-1. Dostupný z WWW: <http://www.root.cz/clanky/jan-minarik-s-rails-vyvijime-efektivneji/>. ISSN 1212-8309.

[15] HANSSON, David. Weblog D.H.Hanssona [online]. 2007 [cit. 2008-03-10]. XHTML 1.0 Transitional. Angličtina. Dostupný z WWW: <http://www.loudthinking.com/about.html>.

[17] ŠŤASTNÝ, Jakub. Bylo vydáno JRuby 1.0. Root.cz [online]. 2007 [cit. 2008-03-14], s. 1-1. Dostupný z WWW: <http://www.root.cz/zpravicky/bylo-vydano-jruby-1-0/>.

[18] SLOMINSKI, Aleksander. Ruby\'s Type Explosion aka Duck Typing. Alek Links Etc. [online]. 2005 [cit. 2008-03-18], s. 1-1. Dostupný z WWW: <http://alek.xspaces.org/2005/02/27/ruby-type-explosion>.

[19] ŠRÁMEK, Dalibor. Ruby a OOP. Root.cz [online]. 2002 [cit. 2008-04-16], s. 1-1. Dostupný z WWW: <http://www.root.cz/clanky/ruby-a-oop-3/>. ISSN ISSN 1212-830.

Kapitola 10. Příloha CD - seznam a uspořádání přikládaného CD

10.1. Adresářová struktura

```
codes
     SudokuWeb
database
images
literatura
oldcodes
     Sudoku
     SudokuWeb
prezentace
     images
prirucka
     images
samples
     Kolekce
     pokus
     scaffold
tools
     InstantRails
            InstantRails1.7
            InstantRails2.0
      NetBeans
bakalarka.pdf
bakalarka.xml
```

codes/SudokuWeb

Zde najdete kompletní zdrojové kódy pro aplikace webového serveru Sudoku. Celý tento adresář odpovídá struktuře Rails projektu. Pro spuštění stačí spustit Ruby (např. server z InstantRails), spustit server WEB-rick pomocí `ruby script/server` v tomto adresáři aplikace a v prohlížeči zadat adresu: `http://localhost:3000/main`.

database

Vyexportovaná databáze MySQL. V této databázi se nachází sbírka úloh sudoku určených k řešení pro uživatele. Najdete zde i naplněnou tabulku users několika uživateli. Databáze je určena pro přímý import na server a spuštění aplikace.

images

Tento adresář obsahuje obrazové materiály, které byly použity při generování tohoto kompletního PDF dokumentu bakalářské práce.

literatura

Adresář obsahuje tři PDF dokumenty, ze kterých jsem volně čerpal při psaní mé bakalářské práce. Jedná se většinou o popisy sudoku problému.

oldcodes/Sudoku

Zde se nachází původní zdrojové kódy pro konzolovou aplikaci Sudoku. Jedná se o můj první nástřel funkčnosti sudoku a jeho obsluhy. V této aplikaci pouze testuji algoritmy nutné k řešení sudoku. Tyto prvotní kódy jsou samozřejmě součástí mé práce i samotného vývoje finální aplikace.

oldcodes/SudokuWeb

Zdrojové kódy projektu webového serveru sudoku před očištěním od různých testovacích metod a před vypuštěním nepoužívaných bloků kódu. Očištěním tohoto vývojového projektu vznikla finální aplikace.

prezentace/images

Obrazový materiál, který byl použit při tvorbě prezentace mé bakalářské práce.

prezentace

Soubory Microsoft PowerPoint určené k prezentaci práce.

prirucka/images

Obrázky a screenshoty pro uživatelskou příručku.

prirucka

Uživatelská příručka: XML podoba i výsledný soubor PDF.

samples/Kolekce

Ukázková aplikace pro práci s kolekcemi v Ruby.

samples/pokus

Soubor `task.rb` obsahuje kód ukázkového příkladu zpracování Stringů v Ruby (jejich převod na Integer).

samples/scaffold

Rails projekt ukázkového příkladu na použití pomocného nástroje scaffold.

tools/InstantRails

Pro všechny, kteří chtějí začít okamžitě s Ruby on Rails. Ideální startovací nástroj InstantRails pro vývoj. Umístil jsem zde dvě verze: Rails1.7 pro zpětnou kompatibilitu (v této verzi Rails

jsem vyvíjel i svou aplikaci) a novou verzi Rails2.0, která ale bohužel již není zpětně kompatibilní se staršími verzemi.

NetBeans

Instalační soubor vývojového prostředí NetBeans IDE 6.01. Nejlepší nástroj pro vývoj v Ruby on Rails v současné době.

bakalarka.pdf

Tento PDF dokument bakalářské práce. Vygenerováno nástrojem XEP.

bakalarka.xml

Zdrojový soubor pro PDF dokument. Použita šablona Docbook ve formátu XML.