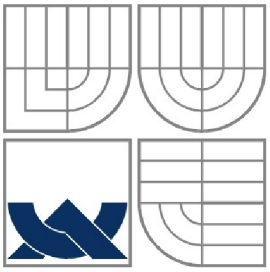
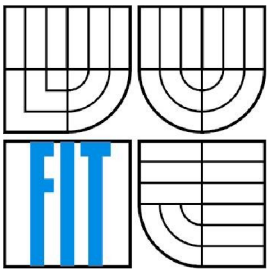


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# UNIVERZÁLNÍ SYSTÉM PRO SPRÁVU OBSAHU V RUBY ON RAILS

UNIVERSAL CONTENT MANAGEMENT SYSTEM IN RUBY ON RAILS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE
AUTHOR

JAN KORIŤÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. DUŠAN VRÁŽEL

BRNO 2008

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav informačních systémů Akademický rok 2007/2008

# Zadání bakalářské práce

Řešitel: **Koriťák Jan**
Obor: Informační technologie
Téma: **Univerzální systém pro správu obsahu v Ruby on Rails**
Kategorie: Web

Pokyny:

1. Seznamte se s programovacím jazykem Ruby a rámcem Ruby on Rails.
2. Seznamte se s existujícími systémy pro správu obsahu a požadavky kladenými na tyto systémy.
3. Navrhněte univerzální systém pro správu obsahu pro rámec Ruby on Rails. Při návrhu využijte jazyka UML. Rozsah a funkčnost systému konzultujte s vedoucím.
4. Systém implementujte a ověřte funkčnost systému vytvořením minimálně třech odlišných webových prezentací. Rozsah a druh webových prezentací konzultujte s vedoucím.
5. Zhodnoťte dosažené výsledky, zejména porovnejte vaše řešení s již existujícími a zhodnoťte užití rámce Ruby on Rails.

Literatura:

- Informace a dokumentace dostupné na http://www.**ruby**on**rails**.org/.
- Thomas D., Hansson H.: Agile Web Development with Rails: Second Edition. Pragmatic Bookshelf 2006.

Při obhajobě semestrální části projektu je požadováno:

- Body 1. až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Vrážel Dušan, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Ing. Jaroslav Zendulka, CSc.
*vedoucí ústavu*

# LICENČNÍ SMLOUVA
# POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

**1. Pan**

Jméno a příjmení: **Jan Koriťák**
Id studenta: 79336
Bytem: Zbýšov 56, 285 65 Zbýšov v Čechách
Narozen: 24. 01. 1986, Městec Králové
(dále jen "autor")

a

**2. Vysoké učení technické v Brně**

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

## Článek 1
## Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Univerzální systém pro správu obsahu v Ruby on Rails
Vedoucí/školitel VŠKP: Vrážel Dušan, Ing.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

## Článek 2
## Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

## Článek 3
## Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

| | |
|---|---|
| ..... | ..... |
| Nabyvatel | Autor |

## Abstrakt

Práce se zabývá použitím webových frameworků při vývoji internetových aplikací. Popisuje systémy pro správu obsahu a požadavky na ně kladené. Srovnává vybrané zástupce webových frameworků a systémů pro správu obsahu. Cílem práce je implementace vlastního systému pro správu obsahu v prostředí frameworku Ruby on Rails. Systém má umožňovat snadnou publikaci dokumentů rozličných forem, podporovat jazykové mutace a umožňovat použití vlastních šablon vzhledu.



## Klíčová slova

Web, framework, systém pro správu obsahu, Ruby, Ruby on Rails, Model-Pohled-Řadič



## Abstract

This paper introduces the reader into utilization of web application frameworks in web development. It describes Content Management Systems and demands on them. The goal of the thesis is to implement a new Content Management System using a Ruby on Rails framework. The system is required to faciliate a simple publication of documents in various forms. It should also provide multilingual environment and support custom interface templates.



## Keywords

Web, framework, Content Management System, Ruby, Ruby on Rails, Model-View-Controller



## Citace

Jan Koriťák, Univerzální publikační systém v Ruby on Rails, bakalářská práce, Brno, FIT VUT v Brně, 2008

# Univerzální systém pro správu obsahu v Ruby on Rails

## Prohlášení

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

.....
Jan Koriťák
12. května 2008

## Poděkování

Rád bych poděkoval Ing. Dušanu Vráželovi za vedení a rady při tvorbě této práce.

# Obsah

# 1 Úvod

Vývoj webových aplikací při použití běžných postupů spočívá ve vytváření sady programů ve skriptovacích jazycích, které generují kód webových stránek na základě údajů v databázi. Pokud s každou novou aplikací začínáme na této sadě pracovat znovu od začátku, přináší to nezanedbatelné časové nároky a hlavně často vede k nutnosti znovu vytvářet některé programové části, které jsou pro množství aplikací společné a byly již dříve mnohokrát implementovány.

Při budování webového sídla tradičními metodami se nezanedbatelnou část stráveného času vůbec nezabýváme samotnou realizací funkcí specifických pro naší aplikaci a její účel, nýbrž různými podpůrnými rutinami (např. mechanismus vykreslování stránky, předávání dat mezi moduly aplikace a přihlašování uživatelů).

Jedním z řešením je použití frameworků pro webové aplikace, které se snaží popsané problémy eliminovat. Je-li zamýšlená aplikace založena pouze na principech publikace a manipulace s dokumenty, stojí za úvahu programování úplně vypustit a použít již hotový systém pro správu obsahu, který může nabízet možnosti přizpůsobení do námi přesně požadované formy.

Předmětem této práce je realizace vlastního systému pro správu obsahu s využitím frameworku Ruby on Rails.

Kapitola 2 popisuje používané pojmy a dotčené technologie. Vysvětluje mj. principy frameworků a systémů pro správu obsahu. Dále jsou v ní rozebrány a srovnány některé dostupné produkty. Zvláštní pozornost je věnována jazyku Ruby a frameworku Ruby on Rails.

V kapitole 3 jsou specifikovány požadavky na vlastní systém pro správu obsahu podle zadání této bakalářské práce. Návrh systému je proveden v kapitole 4 a kapitola 5 se týká vlastní implementace systému.

## 1.1 Typografické konvence

V textu budou dodržována následující pravidla:

Názvy modulů, tříd a konstant budou v textu zvýrazněny neproporcionálním písmem, např. `ActiveRecord`.

Jiné identifikátory a ukázky kódu jsou vysázeny neproporcionálním písmem odlišného typu, např. `self.validate()`.

Kurzívou jsou vysázeny nově zaváděné pojmy, názvy tabulek a jiné speciální řetězce použité v programu, např. *pohled*, *DocumentList*.

# 2 Technologie a pojmy

## 2.1 Skriptovací jazyky a Web

### 2.1.1 Motivace

V dnešní době se již málokdo spokojí s jednoduchou webovou prezentací se statickým obsahem. U webových aplikací se klade důraz na dynamičnost – požadujeme aby obsah, který zobrazují, odrážel aktuální skutečnosti a reagoval na jejich změny. K dosažení tohoto cíle si nevystačíme s prostým jazykem pro formátování textu, jako je HTML.

Webové stránky už dávno nejsou jen pouhé dokumenty. Staly se z nich programy, které tyto dokumenty dynamicky generují zejména na základě dat v databázi. Tyto vygenerované dokumenty, které se přenesou klientovi ke zpracování internetovým prohlížečem, mohou obsahovat další programy, které se spustí až na počítači klienta, a mohou dokument dále upravovat.

Zde se tedy setkáváme s pojmy client-side skriptování a server-side skriptování. Client-side skriptování slouží k programování chování dokumentu na straně internetového prohlížeče a má zejména prezentační účel. S tématem této práce však více souvisí druhý zmíněný pojem. Server-side skriptování se používá k dynamické přípravě informací, ze kterých se generuje výsledný dokument. Dále může sloužit ke spouštění jiných akcí, jako jsou např. úpravy dat v databázi, či automatické odesílání e-mailových zpráv.

### 2.1.2 Server-side skriptování

Server-side skriptování je technologie používaná na webových serverech. Spočívá ve zpracování požadavku od uživatele skriptem (obslužným programem) a dynamickém vygenerování HTML stránek. Většinou se používá k realizaci interaktivních webových aplikací, které jsou napojeny na databázi.

Původně bylo server-side skriptování realizováno programy v jazyce C, Perl a shellovými skripty s využitím rozhraní Common Gateway Interface (CGI). Tyto programy byly vykonány operačním systémem na požadavek webového serveru, kterému se po jejich vykonání pouze předal výsledek k odeslání klientovi (vygenerovaný HTML dokument).

V dnešní mohou být programy v některých jazycích vykonány přímo webovým serverem, nebo jeho rozšiřujícím modulem.

Pomocí libovolné z těchto forem skriptování (tzn. CGI, nebo přímé vykonání serverem) lze vytvořit komplexní webové aplikace, nicméně přímé vykonání serverem povětšinou představuje nižší režii díky absenci volání externích interpretů skriptovacích jazyků.

Při popisu server-side skriptování bylo čerpáno z [1].

## 2.2 Ruby

Ruby je dynamický, reflexivní, objektově orientovaný programovací jazyk kombinující syntax inspirovanou jazykem Perl a rysy podobné jazyku Smalltalk. Tvůrcem jazyka Ruby je japonský programátor Yukihiro Matsumoto, který začal s prací na jazyce v roce 1993 a v roce 1995 jej uvolnil.

Na [2] lze najít mnoho podrobnějších informací o vývoji a implementacích jazyka Ruby.

Ruby je čistě objektový jazyk. Veškeré proměnné jsou tedy vždy objekty odvozené od kořenového objektu `Object`. Dokonce i jednotlivé třídy a moduly jsou ve skutečnosti odvozené od objektu `Object`.

V následujících kapitolách se pokusím nastínit základy syntaxe jazyka Ruby a popsat jeho prvky, které jej odlišují od jiných běžně známých jazyků. Text bude mít spíše popularizační charakter, neboť účelem této práce není vyčerpávající popis všech aspektů a konstrukcí, které jazyk nabízí. Případný zájemce najde dostatek informací například v dokumentaci k Ruby na webové adrese [3], nebo v publikaci [4].

### 2.2.1 Základy syntaxe

V jazyce Ruby konvence pro pojmenování proměnných nahrazuje modifikátory rozsahu platnosti známé z jiných jazyků. Identifikátory konstant a tříd musí začínat velkým písmenem. Identifikátory začínající malým písmenem jsou lokální proměnné.

Názvy takzvaných instančních proměnných – atributů objektu sdílených všemi metodami objektu – musí začínat znakem '`@`'. Názvy atributů třídy (jinak známých jako statické proměnné) musí začínat řetězcem '`@@`'. Názvy globálních proměnných začínají znakem '`$`'.

Atributy objektů i tříd jsou viditelné pouze v rámci objektu (či třídy). Neexistuje žádný modifikátor, který by zajistil jejich viditelnost zvenčí. Jedinou možností je tedy implementace přístupových metod k těmto atributům. Tím je vlastně programátor nucen psát kvalitnější a srozumitelnější kód.

Zajímavostí je, že názvy metod mohou končit neobvyklými znaky '?' a '!'. Těchto znaků se s oblibou používá k naznačení smyslu metody. Metoda s názvem končícím otazníkem většinou vrací logickou hodnotu a neprovádí změny v objektu. Končí-li název metody vykřičníkem, jedná se obvykle o metodu, která má nějaký nevratný efekt. Metoda může končit i znakem '=', taková metoda je potom

volána formou přiřazení. To se většinou používá k zpřístupnění vnitřních atributů pro zápis vnějšími objekty.

Základní prvky syntaxe jazyka Ruby jsou předvedeny na *příkladu 1*:

**Příklad 1**:

```
class Clovek
  @@pocet = 0
  def Clovek.pocet
    @@pocet
  end

  def initialize
    @@pocet += 1
    poradi = @@pocet
  end

  def poradi
    @poradi
  end

  protected

  def poradi=(hodnota)
    @poradi = hodnota
    puts "Přiřazeno pořadové číslo #{hodnota}."
  end
end
```

V tomto jednoduchém příkladu byla vytvořena třída `Clovek` s atributem `@@pocet` udávajícím počet dosud vytvořených objektů této třídy. Objekty této třídy obsahují atribut `@poradi` který představuje informaci o tom, kolikátý v pořadí byl objekt vytvořen.

Z příkladu tedy vidíme, že třídy se v Ruby definují blokem `class <třída> ... end` a metody blokem `def <metoda>[(parametry)] ... end`.

Konstruktor třídy se zapisuje jako definice metody `initialize` (tato metoda může akceptovat libovolný počet parametrů). Konstruktor v našem příkladu provede inkrementaci atributu třídy určujícího počet vytvořených objektů a dále provede inicializaci pořadového čísla nově vytvářeného objektu.

Povšimněme si však, že pořadové číslo není přiřazeno přímým zápisem do atributu objektu, nýbrž voláním metody `poradi=`, která kromě vlastního přiřazení zároveň vypíše na standardní výstup informaci o přiřazení. Definice metody se nachází pod modifikátorem `protected`, metoda tedy není viditelná z vnějších objektů. Tím je zajištěna nemožnost změny pořadového čísla zvenčí. V metodě si zároveň můžeme všimnout způsobu extrakce výrazů do řetězce speciálním řetězcem `#{}`. Nepřejeme-li si využívat tohoto chování, ať už z důvodu kolize speciálních znaků, nebo optimalizace aplikace, můžeme pro definici řetězcových literálů použít apostrofy místo uvozovek.

Nakonec se podívejme na metodu třídy pojmenovanou `pocet`, která poskytuje přístup ke čtení atributu `@@pocet`. Vidíme, že definice metody třídy se zapisuje explicitním uvedením názvu třídy před názvem metody odděleného tečkou. Místo názvu třídy lze použít také speciální klíčové slovo `self`. Zde si můžeme také všimnout, že v metodách není nezbytné používat příkaz `return`. Není-li příkaz `return`

použit, návratová hodnota metody je automaticky stanovena na hodnotu naposledy vyhodnoceného výrazu.

### 2.2.2 Symbol

Jednou ze zvláštností jazyka Ruby je datový typ Symbol inspirovaný symboly v jazyce LISP. Symbol slouží k reprezentaci programátorem definovaných názvů. Podobá se řetězcovému datovému typu, má však několik důležitých odlišností. Název symbolu nelze upravovat žádnými funkcemi (lze jej v případě potřeby převést na String, který lze dále upravovat, objekt typu Symbol však zůstane nezměněn). Symboly se stejnými názvy představují totožný objekt (na rozdíl od řetězců – řetězce se stejným obsahem jsou různé objekty). Při použití symbolů tedy dochází k úspoře paměti. Vnitřní reprezentace symbolu je číselná, s výhodou tedy můžeme používat symboly jako klíče v kolekcích typu Hash (bude popsáno v kapitole 2.2.4). Symbol se zapisuje jako identifikátor začínající dvojtečkou. *Příklad 3* ukazuje použití symbolu jako klíče v kolekci typu Hash.

V případě potřeby máme možnost převést symbol na řetězec metodou `to_s` (např. `:brno.to_s` vrátí `"brno"`), a naopak vytvořit symbol z řetězce metodou `intern` (např. `"brno".intern` vrátí `:brno`).

### 2.2.3 Pole

Každá proměnná typu pole je v Ruby objektem třídy Array. Tato třída poskytuje řadu užitečných metod pro práci s polem, které lze nalézt na [3]. Při vytváření pole není nutné zapisovat vytváření nového objektu, lze využít zjednodušeného inicializačního zápisu (viz *Příklad 2*).

**Příklad 2**:

```
a = [1, 2, 3]
b = []
```

Tento zápis vytvořil pole `a` se třemi prvky a prázdné pole `b`. Pole se indexuje od nuly pomocí hranatých závorek. První hodnotu v poli `a` získáme tedy zápisem `a[0]`.

### 2.2.4 Hash

Pojmem hash se v Ruby rozumí datová struktura jinak nazývána asociativní pole, případně slovník. Hashe jsou objekty třídy `Hash`, která obsahuje řadu užitečných funkcí pro práci s hashem. Jejich popis lze nalézt na [3]. Označení hash pro tuto strukturu může být pro českého čtenáře mírně zavádějící, zvolil jsem jej však z důvodu stejného použití pojmu v zatím jediné dostupné literatuře v českém jazyce - [5].

Jedná se o strukturu podobnou poli, ve které je k uloženým prvkům místo indexování číslem přistupováno pomocí klíče, který může tvořit libovolný objekt.

I pro objekty třídy Hash existuje zjednodušený inicializační zápis (viz *Příklad 3*).

**Příklad 3**:

```
a = { :klic => 'hodnota', :klic2 => 'hodnota2' }
b = {}
```

Přistupování k prvkům ve struktuře se provádí stejně jako u pole, tj. hranatými závorkami. Hodnotu v hashi `a` pod klíčem `:klic` tedy získáme zápisem `a[:klic]`.

Zajímavostí je, že při volání metody, kdy jako poslední parametr předáváme hash, můžeme vynechat složené závorky. Oba způsoby volání ukazuje *příklad 4*. Tímto způsobem lze nahradit funkci pojmenovaných parametrů, kterou mají některé jazyky.

**Příklad 4**:

```
metoda(parametr, {:klic => 'hodnota', :klic2 => 'druha_hodnota'})
metoda(parametr, :klic => 'hodnota', :klic2 => 'druha_hodnota')
```

### 2.2.5 Bloky

Ruby umožňuje předávat bloky v parametrech metod. To lze využít k psaní obecných metod, jejichž chování je ovlivněno v místě volání. K provedení bloku, který byl předán jako parametr metody slouží příkaz `yield`. Tento příkaz také akceptuje libovolný počet parametrů, které jsou pak prováděnému bloku předány. Návratová hodnota příkazu odpovídá návratové hodnotě bloku.

Předávání bloku metodě se provádí připojením bloku za volání metody. Blok je ohraničen buď složenými závorkami, nebo klíčovými slovy `do` a `end`. Na začátku bloku může být seznam parametrů, které blok akceptuje. Tento seznam se zapisuje do svislých čar a jednotlivé parametry se oddělují čárkami.

*Příklad 5* ukazuje definici a volání metody `zabezpeceno`, která provede předaný blok pouze za předpokladu, že je uživatel programu přihlášen. V takovém případě je navíc bloku předán parametr se jménem tohoto uživatele.

Není-li uživatel přihlášen, je zobrazena chybová zpráva, která je předána jako standardní parametr metody.

**Příklad 5**:

```
def zabezpeceno(chybova_zprava)
  if Uzivatel.prihlasen? then
    yield Uzivatel.jmeno
  else
    puts chybova_zprava
  end
end

zabezpeceno('Operace vyžaduje přihlášení') do |jmeno_uzivatele|
  Platby.odeslat!
  Zaznamy.pridat("Uživatel #{jmeno_uzivatele} provedl odeslani plateb")
end
```

Předávání bloku je velmi často využíváno při volání metody `each`, která je obsažená v polích a hashích. Tato metoda provede se všemi prvky kolekce operaci definovanou blokem. Blok je tedy volán znovu pro každý prvek v kolekci. Tento prvek je vždy předán bloku jako jeho parametr.

Metoda `each` nahrazuje cyklus typu `for each` známý z jiných jazyků. Použití ukazuje *příklad 6*. Ukázkový kód představuje průchod polem objektů reprezentujících zákazníky a vypsání zákazníků, kteří mají nějaké nesplacené pohledávky.

**Příklad 6**:

```
zakaznici.each do |zakaznik|
  if zakaznik.pohledavka > 0 then
    puts "Dlužník: #{zakaznik.jmeno}, #{zakaznik.pohledavka} Kč"
  end
end
```

### 2.2.6 Moduly

Dalším specifikem jazyka Ruby jsou moduly. Modul je kolekce metod, konstant a vnořených modulů podobně jako třída, nelze jej však instanciovat. Moduly v jazyce Ruby nahrazují mechanismy jmenných prostorů, vícenásobné dědičnosti a také se používají při reflexivním programování pro rozšiřování tříd.

Třída je v jazyce Ruby vlastně speciální případ modulu, který má schopnost vytvářet instance.

Modul použijeme, chceme-li definovat sadu metod, které k sobě sice logicky patří, není však třeba, aby pracovaly s nějakým objektem. Příkladem může být modul matematických funkcí. Řekněme, že potřebujeme obohatit náš program o sadu speciálních matematických metod. Definujeme tedy modul `SpecialMath`, ve kterém budeme definovat metody modulu, např. `integrate_newton`, `solve_equation` apod. Tím, že jsme umístili metody do modulu, můžeme kdekoliv v aplikaci např. zápisem `SpecialMath.SolveEquation(...)` provést danou matematickou operaci. Stejný efekt by se dal dosáhnout použitím třídy a třídních metod, nicméně v tomto případě není důvod dávat matematickému modulu možnost vytvářet instance.

Modul má ale i další uplatnění. Každý modul v Ruby (a tedy i každá třída) definuje metodu `include`. Touto metodou je možné přidat konstanty, metody a proměnné modulu uvedeného v parametru metody do modulu přijímajícího volání metody. Častou praxí při implementacích rozšiřujících kódů aplikací či zásuvných modulů je vytváření modulů s novými funkcemi a jejich vkládání do existujících tříd. Tento případ je ukázán na *příkladu 7*.

**Příklad 7**:

Mějme třídu pro objednávky, která obsahuje metodu pro výpočet celkové ceny za zboží v objednávce součtem koncových cen všech produktů, které se zjistí vynásobením koeficientem provize:

```
class Objednavka
  def cena_celkem
    mezisoucet = 0
```

```
    produkty.each do |produkt|
      mezisoucet += koncova_cena(produkt)
    end
    mezisoucet
  end

  def koncova_cena(produkt)
    produkt.cena * provize
  end
end
```

Uvažujme že potřebujeme realizovat rozšíření, které pro produkty patřící do dané kategorie bude koncovou cenu počítat vynásobením jiným koeficientem provize. Znovu nadefinujeme metodu `koncova_cena` s novým chováním a umístíme jí do modulu `ObjednavkaRozsireni`.

```
module ObjednavkaRozsireni
  def koncova_cena(produkt)
    if (produkt.kategorie == :lepsi_produkty) then
      return produkt.cena * vetsi_provize
    else
      return produkt.cena * provize
    end
  end
end
```

Nakonec provedeme vložení modulu `ObjednavkaRozsireni` do třídy `Objednavka`. Metody, konstanty a proměnné třídy zůstanou beze změny, změní se jen upravená metoda `koncova_cena`. Všimněme si, že jazyk Ruby nám umožňuje kdykoliv „znovuotevřít“ jakoukoliv třídu (nebo modul) a dodefinovat nové metody, překrýt již existující, nebo dodatečně vložit modul s novými metodami.

```
class Objednavka
  include ObjednavkaRozsireni
end
```

## 2.3 Frameworky

Framework pro webové aplikace je sbírka knihoven podporujících vývoj dynamických webových aplikací. Jeho účelem je snížit časovou režii při vývoji webových aplikací zapouzdřením běžně používaných postupů. Množství frameworků tedy poskytuje nástroje pro přístup k databázi, šablonovací systémy a mechanismy pro udržování sezení. Frameworky často prosazují znovupoužitelnost kódu.

Tzv. „full-stack“ frameworky mají několik částí pro podporu různých aspektů vývoje webových aplikací, které jsou sloučeny do jednoho softwarového balíku. Příkladem mohou být frameworky JavaEE, OpenACS a Ruby on Rails.

### 2.3.1 Architektura Model-View-Controller

Architektura, kterou používá množství webových frameworků včetně Ruby on Rails, se nazývá Model-View-Controller (zkráceně MVC). Tato architektura slouží k vzájemnému oddělení logiky aplikace, da-

tového modelu a uživatelského rozhraní. Její použití přispívá ke kvalitě softwarových produktů a urychluje vývoj.

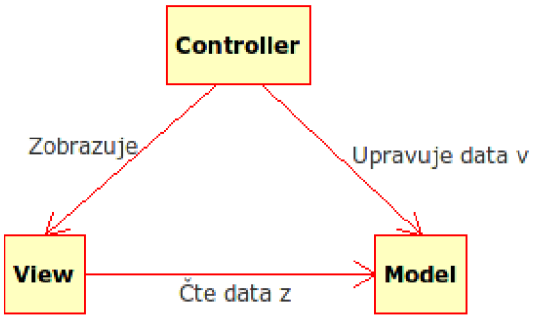
Architektura se skládá ze tří komponent. *Modely* jsou třídy sloužící k datové abstrakci. Každý model reprezentuje entitu reálného světa, má její atributy a poskytuje požadované operace nad ní. Implementovaný mechanismus MVC potom může zajišťovat perzistenci modelů. Operace nad modelem jsou většinou převáděny na dotazy databázovému serveru.

*Pohledy* (views) se starají o reprezentaci dat získaných z modelů. Ve webových aplikacích se většinou jedná o šablony HTML dokumentů, které obsahují značky pro vkládání dynamických dat. Pohledy tedy slouží k interakci s uživatelem a představují uživatelské rozhraní aplikace.

*Řadiče* (controllers) obsahují řídící logiku aplikace.

Fungování aplikace v MVC architektuře tedy probíhá tak, že podle druhu uživatelova požadavku je zvolen řadič, který tento požadavek zpracuje, volitelně provede operace s modely a zvolí pohled, který načte potřebná data z modelů a zobrazí se uživateli.

Spolupráci komponent ilustruje obrázek 1.




*Obrázek 1: Spolupráce komponent architektury MVC*

### 2.3.2 Dostupné produkty

Frameworků pro vývoj webových aplikací existuje celá řada. Při volbě frameworku se většinou řídíme použitým programovacím jazykem, dostupností dokumentace, aktivitou vývojářů frameworku, poptávkou na trhu práce a výkonem vytvořených aplikací.

V současné době je asi nejpopulárnějším jazykem používaným pro vývoj webových aplikací jazyk PHP. Je velmi jednoduchý na naučení a po vývojářích v PHP je na trhu práce vysoká poptávka. Nicméně kvality produktů napsaných v PHP jsou velmi rozdílné. Jazyk PHP může hlavně začínající programátory svádět k psaní velmi nekvalitních zdrojových kódů, jejichž špatná srozumitelnost, rozšiřitelnost a znovupoužitelnost vede k selhávání větších projektů, které s přibývajícími požadavky přiná-

šejí stále větší režii na práci vývojářů. To však neznamená, že v PHP nelze kvalitně implementovat rozsáhlý projekt. Vše záleží na zkušenostech a přístupu vývojářů.

Pro jazyk PHP existuje řada frameworků pro vývoj webových aplikací. Mezi nejznámější patří CodeIgniter (poslední verze 1.6.1. z 12. února 2008, viz [6]) a CakePHP (poslední stabilní verze 1.1.19 z 1. ledna 2008, viz [7]). Oba frameworky mají slušnou dokumentaci a zázemí v aktivní komunitě uživatelů.

V internetových diskuzích velmi chváleným je framework Django (poslední verze 0.96.1, viz [8]). Framework je pro jazyk Python, který má také vysoký počet uživatelů. Frameworku Django je připisován slušný výkon na základě různých srovnávacích testů (např. [9]). V současné době je však jeho nevýhodou absence stabilní verze. Před vydáním stabilní verze lze tedy očekávat razantní změny v API rozhraní, což může způsobit problém aplikacím vytvořených pro vývojovou verzi frameworku. Poptávka po vývojářích v Djangu u nás je momentálně mizivá.

Zajímavou volbou je framework Ruby on Rails (poslední verze 2.0 ze 17. prosince 2007, viz [10]), na který je zaměřena tato práce. Jak už název napovídá, použitým jazykem je Ruby. Ruby on Rails se může pochlubit silnou komunitou, dostupnou literaturou a kvalitní dokumentací. V Ruby on Rails byla vytvořena řada komplexních webových aplikací, jako je např. Basecamp - internetová aplikace pro projektový management, která má přes 1 000 000 uživatelů (více na [11]). Specialitou Ruby on Rails je snadné používání technologie AJAX pro pokročilé prezentační prvky. Poptávka po vývojářích v Rails je u nás minimální.

## 2.4 Ruby on Rails

Ruby on Rails je volně dostupný framework pro vývoj webových aplikací vytvořený pro rychlejší, jednodušší a efektivnější vývoj.

Vytvořil ho David Heinemeier Hansson při práci na projektu Basecamp (viz [11]) ve společnosti 37signals ([12]).

Mezi základní principy frameworku Ruby on Rails patří upřednostňování konvence před konfigurací a DRY (Don't repeat yourself – neopakujte se).

Pojem upřednostňování konvence před konfigurací znamená, že vývojář musí specifikovat pouze nekonvenční aspekty aplikace. Například, existuje li třída `Document` v modelu (podle MVC, viz 2.3.1), odpovídající tabulka v databázi je implicitně pojmenována *documents*. Pouze má-li vývojář v úmyslu se od konvence pojmenování odchýlit, musí dopsat kód pro vazbu třídy na tabulku.

Pojem DRY představuje snahu o umístění každé informace na jednom jednoznačném místě bez potřeby opakovat ji na jiném místě.

Jako většina současných webových frameworků, i Rails jsou postaveny na architektuře MVC (viz 2.3.1).

Za zmínku stojí obsáhlé využívání JavaScriptových knihoven Prototype a Script.aculo.us pro AJAX a efekty grafického rozhraní. Navíc je podporován mechanismus REST pro webové služby. Těmito součástmi se tato práce nebude zabývat. Více informací o JavaScriptových knihovnách v Rails lze nalézt například na [13], více informací o mechanismu REST pak například na [14].

Pro studium prostředí Ruby on Rails existuje v českém jazyce publikace [5]. Pro lepší proniknutí a porozumění frameworku lze však doporučit spíše anglicky psanou knihu [15].

### 2.4.1 Struktura frameworku

Ruby on Rails se skládá z několika různých balíčků, konkrétně ActiveRecord, ActiveResource, ActionPack, ActiveSupport a ActionMailer. Kromě těchto balíků mohou vývojáři přispívat vlastními zásuvnými moduly s rozšířením těchto balíčků či jinými rozšířeními.

Stěžejními součástmi jsou balíčky ActiveRecord a ActionPack, kterými se budou dále zabývat příslušné kapitoly. Balíček ActiveResource slouží k využívání mechanismu REST pro implementaci webových služeb, ActionMailer je balíček pro odesílání e-mailů z aplikace a ActiveSupport je kolekcí podpůrných funkcí a rozšíření jazyka Ruby.

### 2.4.2 ActiveRecord

ActiveRecord je třída pro specifikaci modelů (z pohledu MVC, viz 2.3.1). Implementuje mechanismus objektově-relačního mapovaní (ORM).

Objektově-relační mapování spočívá v reprezentaci řádků tabulek v databázi objekty. Programátor nemusí k přístupu k datům vůbec používat SQL dotazy, stačí používat metody modelových objektů a tříd.

ActiveRecord ctí princip Ruby on Rails upřednostňování konvence před konfigurací. Používá například metody automatické pluralizace pro odhad jména tabulky v databázi podle názvu třídy implementující model. Tato automatická pluralizace funguje jen v anglickém jazyce, je proto doporučeno při vývoji používat identifikátory a názvy tabulek v anglickém jazyce.

Ukázky práce s ActiveRecordem včetně porovnání s často používaným postupem ručního vykonávání SQL dotazů v PHP lze nalézt v *příkladu 8*.

**Příklad 8**:

Řekněme, že máme v databázi tabulku *customers* (zákazníci) se sloupci *id* (primární klíč), *fist_name* (jméno) a *surname* (příjmení). Model popisující data v této tabulce realizujeme pouhým vytvořením nové třídy odvozené ze třídy `ActiveRecord`. Podle názvu třídy se automaticky odvodí název tabulky v databázi. Z tabulky jsou automaticky načteny všechny sloupce a podle nich jsou vytvořeny metody a atributy instancí nové třídy.

```
class Customer < ActiveRecord::Base
end
```

Výše uvedený zápis stačí k tomu, abychom mohli plně využívat objektově relačního mapování pro tabulku *customers*. Můžeme například jednoduše vytvořit nový řádek představující nového zákazníka:

```
new_customer = Customer.create(
                        :first_name => 'Jan',
                        :surname => 'Novak'
                )
```

Zavoláním metody `create` třídy `Customer` jsme vytvořili nový objekt `new_customer` třídy `Customer`, který má atributy odpovídající nově vznikajícímu řádku v tabulce.

Objekty tříd zděděných z ActiveRecord poskytují mj. všechny základní operace typu úpravy atributů a odstranění záznamu.

Další užitečnou vlastností ActiveRecordu je snadná práce s asociacemi. Chceme-li např. definovat vazbu 1:N mezi zákazníky a jejich objednávkami (např. v tabulce *orders*), použijeme k tomu makra třídy `ActiveRecord` `has_many` (má mnoho) a `belongs_to` (patří k):

```
class Customer < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

Všimněme si pojmenování maker a symbolů odpovídajícího přirozenému anglickému jazyku. U makra `has_many` je použita pluralizace symbolu definující asociovanou třídu, přičemž u makra `belongs_to` je použité jednotné číslo.

Nalezení zákazníka podle primárního klíče v proměnné `id` a průchod všemi jeho objednávkami by se implementoval následovně:

```
customer = Customer.find(id)
customer.orders.each do |order|
        ...
end
```

Stejného cíle bychom v PHP bez použití objektově-relačního mapování dosáhli takto:

```
$id = intval($id);
$result = mysql_query("SELECT * FROM customers
                              INNER JOIN orders
                              ON orders.customers_id = customers.id
                              WHERE customers.id = $id");
while ($row = mysql_fetch_object($result)) {
        ...
}
```

### 2.4.3 ActionPack

Balík ActionPack se skládá ze dvou částí – ActionView a ActionController. Z pohledu architektury MVC (viz 2.3.1) ActionView slouží k implementaci pohledů a ActionController k implementaci řadičů.

Použití řadičů spočívá ve vytváření tříd řadičů pro jednotlivé části aplikace. Akce, které bude řadič podporovat, se definují veřejnými metodami tohoto řadiče. *Příklad 9* ukazuje použití řadiče pro práci ze zákazníky (customers).

**Příklad 9:**

```
class ApplicationController < ActionController::Base
  # zde se typicky vyskytuje kód společný všem řadičům aplikace
end

class CustomerController < ApplicationController
  def index
    # seznam zákazníků
  end

  def show
    # detail zákazníka
  end

  #další akce

end
```

ActionView je implicitně využíván součástí ActiveController (lze jej nahradit vlastní komponentou implementující pohledy). Instanční proměnné definované v řadičích jsou viditelné i v šablonách ERB (Embedded Ruby – standard pro vkládání bloků programů v Ruby do dokumentů jiných typů, typicky HTML) zpracovávaných komponentou ActionView.

## 2.5 Systémy pro správu obsahu

Systém pro správu obsahu (content management system, zkráceně CMS) je programové vybavení používané k správě a zobrazování dokumentů různé povahy. Webové CMS systémy umožňují snadno publikovat dokumenty osobám bez znalosti technologií používaných pro tvorbu webových stránek.

Většina systémů umožňuje upravování vzhledu uživatelského rozhraní. Díky tomu je možné, aby aplikace vytvořené pomocí takovýchto systémů měly vlastní identitu odlišenou od jiných aplikací využívajících stejný systém.

Kromě dokumentů na textové bázi mohou systémy pro správu obsahu pracovat s obecnými počítačovými soubory, obrazovým a zvukovým materiálem apod. Mnoho společností používá CMS jako primární umístění pro firemní dokumenty, což jim zjednodušuje sdílení dokumentů mezi zaměstnanci.

Některé CMS systémy obsahují tzv „Workflow“ mechanismus sloužící k automatickému transportu elektronického dokumentu mezi uživateli např. za účelem schvalování, přidávání změn apod.

Mezi součásti, kterými CMS systémy typicky disponují, patří:

- Identifikace klíčových uživatelů a jejich rolí
- Schopnost kategorizovat obsah
- Možnost definovat fáze prací na dokumentech
- Sledování a správa více verzí každého dokumentu

Následující kapitoly popisují některé dostupné produkty z oblasti webových systémů pro správu obsahu a snaží se o jejich porovnání v některých aspektech.

### 2.5.1 Mambo CMS

Mambo CMS je vyzrálý projekt open-source redakčního systému pro výstavbu webového sídla. Na [16] lze získat českou verzi systému. Mambo je modulární, lze do něj přidávat různé doplňky, např. doplněk pro vícejazyčnost.

Produkt obsahuje jednoduchou správu oprávnění. Dokumenty lze členit do kategorií a sekcí (2 úrovně kategorizace). Dále obsahuje množství různých nástrojů jako dynamická menu, RSS a automatický rozesílač e-mailů.

Více informací lze nalézt na stránkách projektu ([17]), nebo na [16] - stránce o Mambu v českém jazyce.

### 2.5.2 Drupal

Dalším zajímavým open-source projektem je Drupal. I tento projekt je silně modulární a propracovaný. Administrace a instalace je mírně složitější než u systému Mambo (jistá složitost instalace a nastavení je společnou vlastností většiny CMS systémů – s přibývajícími součástmi a možnostmi roste složitost nastavení).

Mezi výhody Drupalu patří silný šablonovací systém, který pokročilejším uživatelům umožňuje upravit si vzhled své webové aplikace zcela podle svých představ. Propracovaný systém oprávnění v Drupalu lze využít k implementaci aplikace s různými rolemi uživatelů a různými úrovněmi přístupu.

Pomocí Drupalu je rovněž možné vytvořit několik různých aplikací v rámci jedné instalace systému. To může být výhodné z hlediska instalací aktualizací a opravných záplat Drupalu, které stačí

provést jen jednou a projeví se na všech provozovaných aplikacích. Dále je možné sdílet uživatelské účty mezi těmito aplikacemi.

Drupal má kvalitní a dobře dokumentované API, díky čemuž jej lze jednoduše integrovat s jinými řešeními a doplňky.

Více informací lze nalézt na stránkách projektu ([18]), nebo na [19] - stránce o Drupalu v českém jazyce.

# 3 Specifikace a analýza systému

## 3.1 Funkční požadavky

Úkolem je vytvořit systém pro správu obsahu pro univerzální použití. Systém má umožňovat přizpůsobení uživatelského rozhraní pomocí vlastních šablon vzhledu. Dalším požadavkem je přihlašování a správa uživatelů. Vytváření a editace dokumentů by měly být co nejjednodušší, aby s nimi neměl problém ani neznalý uživatel. Předpokládá se možnost vkládání formátovaného textu pomocí ovládacího prvku typu WYSIWIG editor (What You See Is What You Get – Co vidíš, to dostaneš).

Systém bude evidovat oprávnění uživatelů a podle toho dynamicky nabízet takové akce, na které má přihlášený uživatel pravomoci.

V administrační sekci pro provozovatele aplikace vytvořené tímto systémem pro správu obsahu bude k dispozici průvodce specifikací aplikace, kde bude mít provozovatel možnost přizpůsobit si systém podle svých potřeb.

Systém bude podporovat jazykové mutace obsahu a bude umožňovat snadnou editaci dokumentů s ohledem na tyto mutace.

## 3.2 Nefunkční požadavky

Systém bude vytvořen ve frameworku Ruby on Rails. Jako systém pro řízení báze dat bude během vývoje použit MySQL, vzhledem k použití frameworku Ruby on Rails a jeho abstrakce databázového prostředí se však dá předpokládat zaměnitelnost tohoto databázového prostředí jiným prostředím podporovaným frameworkem.

Systém bude možno provozovat na všech platformách s dostupnou implementací jazyka Ruby.

## 3.3 Univerzálnost systému

Univerzálností systému se zde rozumí schopnost realizovat různorodé aplikace založené na uchovávání obsahu. Tyto aplikace by měly mít možnost vlastního, jedinečného vzhledu, což zabezpečí již ze zadání plynoucí vlastnost systému umožňující použití vlastních šablon vzhledu. Dále by mělo být možné realizovat z podstaty odlišné aplikace např. blogy (webové deníčky), firemní stránky a firemní intranet.

K dosažení tohoto cíle je nutné navrhnout nějakou metodu abstrakce takovýchto aplikací, aby bylo možné systém implementovat.

Veškerý obsah zpracovávaný systémem tedy označme pojmem *dokumenty*. Dokumenty budou představovat jednotlivé soubory informací, které budeme chtít zobrazovat. Každému dokumentu přiřadíme množinu *polí*, které budou nabývat nějakých *hodnot*. Například můžeme mít dokumenty představující příspěvky do blogu, které budou mít pole „nadpis“ a pole „text“. Nadpis bude na stránce vzhledově odlišen od textu příspěvku. Dále můžeme mít dokumenty představující reakce na tyto příspěvky, které budou obsahovat pouze pole „text“.

Jelikož lze očekávat, že podobných dokumentů bude v aplikaci více (např. příspěvků do blogu), můžeme jejich společné vlastnosti (pole, která obsahují) shrnout do speciálních entit popisující dokumenty, kterým budeme říkat *druhy dokumentů*.

Dále budeme potřebovat nějakým způsobem realizovat hierarchickou strukturu dokumentů uložených v aplikaci (např. ke každému příspěvku do blogu budeme chtít přiřadit reakce na tento příspěvek). Toho můžeme docílit zavedením speciálního typu pole – množiny poddokumentů. Druhu dokumentu popisujícímu příspěvky do blogu přiřadíme kromě polí „nadpis“ a „text“ další pole „komentáře“, které bude vždy obsahovat množinu poddokumentů. Tyto poddokumenty budou odpovídat druhu dokumentu popisujícímu reakce na příspěvky.

Posledním aspektem, který přispěje k univerzálnosti systému je systém práv uživatelů. Ten umožní definovat operace s dokumentem proveditelné aktuálně přihlášeným uživatelem. Jiné nastavení práv budou mít reakce na příspěvky, které bude moci vytvářet kdokoliv, jiné nastavení budou mít příspěvky do blogu, které bude moci přidávat a upravovat pouze majitel blogu, a jiné nastavení budou mít stránky v intranetu, které nepřihlášeným uživatelům nebudou vůbec viditelné.

# 4 Návrh systému

Univerzální systém pro správu obsahu byl navržen s ohledem na všechny požadavky uvedené v zadání a v kapitole 3. Rozborem navržené podoby systému včetně vysvětlení významu jednotlivých tříd se zabývá kapitola 4.1. Protože systém je celkem složitý, je v kapitole 4.2 uveden příklad spolupráce jednotlivých tříd při realizaci jednoduché aplikace postavené na systému. Navržený šablonovací systém rozebírá kapitola 4.3.

Vzhledem k použití různých mechanismů v Ruby on Rails pro zajištění upřednostňování konvence před konfigurací, jako je například automatická pluralizace názvu tabulky podle názvu třídy modelu v ActiveRecord (viz 2.4.2), jsou veškeré třídy pojmenovány v anglickém jazyce. V zájmu zachování jazykové integrity jsou v anglickém jazyce pojmenovány i metody a atributy těchto tříd.

## 4.1 Struktura tříd v systému

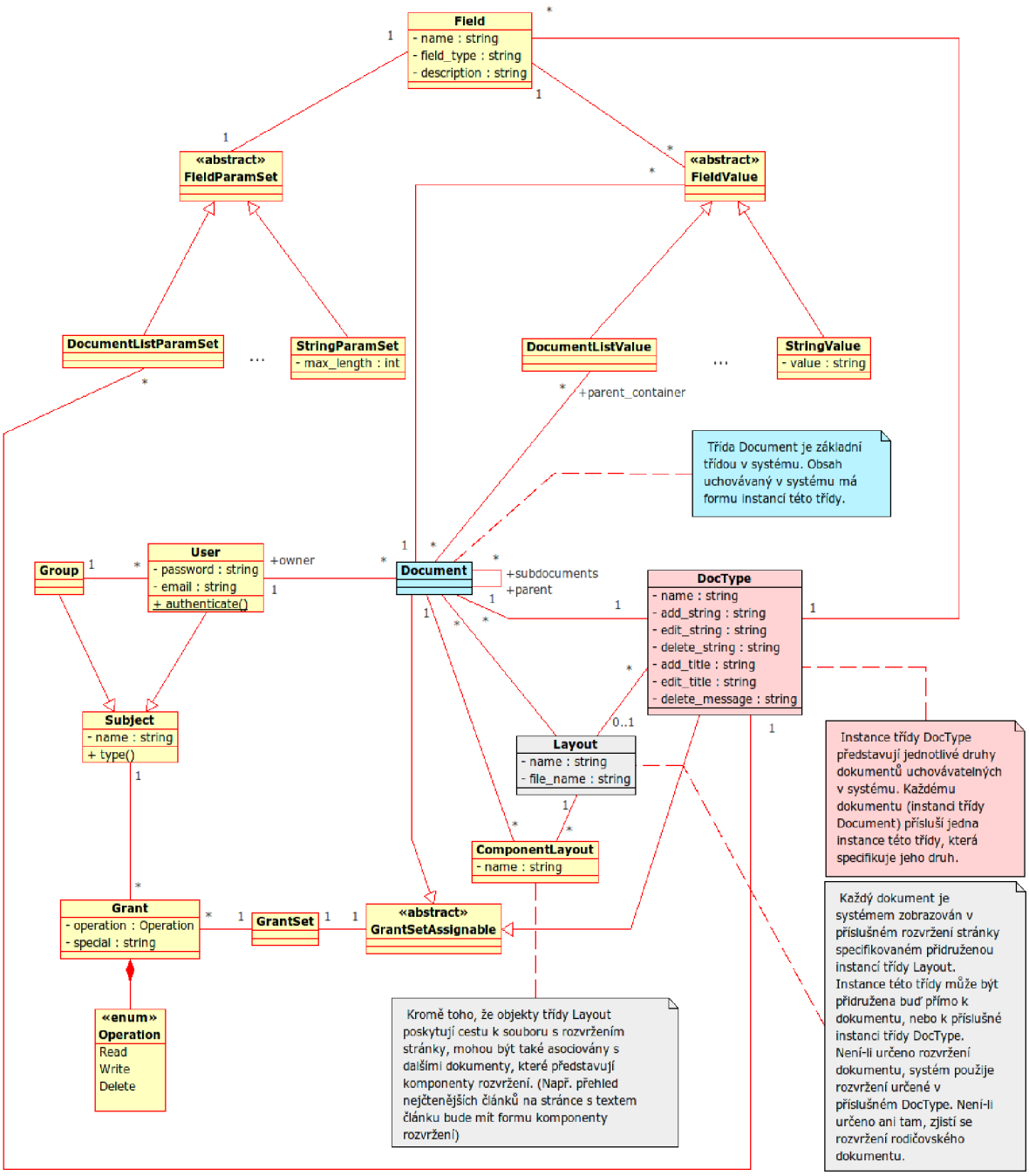
Diagram na *obrázku 2* reprezentuje uspořádání podstatných tříd v systému. Všechny vyobrazené třídy odpovídají modelům architektury MVC (viz kapitola 2.3.1). Jelikož framework Ruby on Rails využívá mechanismu objektově-relačního mapování, každé neabstraktní třídě modelu, která není zděděná z jiného modelu, zároveň odpovídá tabulka v databázi.

V diagramu nejsou v rámci přehlednosti uvedeny metody pro přístup k objektům asociovaných tříd. Tyto metody jsou vždy pojmenovány podle názvu příslušné role v asociaci. Není-li role uvedena, je metoda pojmenována podle názvu asociované třídy. U vícenásobných asociací je použita pluralizace. Název metody je potom převeden na podtržítkovou formu. Dále jsou vynechány metody pro přístup k soukromým atributům. Tyto metody jsou pojmenovány stejně jako odpovídající atributy. Název metod pro přiřazení atributů se skládá z názvu odpovídajícího atributu a znaku '='.

### 4.1.1 Základní třídy

Jádro systému tvoří třídy `Document` a `DocType`. Třída `Document` představuje jednotlivé dokumenty uchovávané v sytému. Dokumentem se zde rozumí univerzální soubor informací, který může mít několik složek různorodého obsahu.

Forma každého dokumentu je definována druhem dokumentu, kterému daný dokument náleží (mechanismus popsán v kapitole 3.3). Proto je ke každému dokumentu přiřazen objekt třídy `DocType`, který specifikuje metadata dokumentu. Pomocí těchto objektů jsou k odpovídajícím dokumentům připojeny další objekty jiných tříd, které se podílí na definici formy dokumentu.




*Obrázek 2: Diagram tříd stěžejní části systému*

Nejdůležitější je připojení objektů třídy `Field`, jenž udávají položky, které daný dokument bude obsahovat. Tyto položky mohou být různých typů a mohou mít různé parametry. Mechanismus položek dokumentů bude popsán dále v kapitole 4.1.4.

### 4.1.2 Rozvržení stránek a šablony

Důležitým aspektem dokumentů spravovaných systémem je jejich vizuální reprezentace. Jedním z požadavků na systém je možnost stanovit každému dokumentu jeho vlastní podobu stránky, která ho bude zobrazovat.

V systému může být nainstalováno několik rozvržení a předpokládá se, že implementátor aplikace postavené na systému bude tvořit vlastní. Každému nainstalovanému rozvržení potom přísluší objekt třídy `Layout`, který uchovává název rozvržení a název adresáře s rozvržením.

S objektem třídy `Layout` mohou být asociovány jednotlivé dokumenty, ale také objekty třídy `DocType`, z kterých se poté jim nastavené rozvržení propaguje na všechny dokumenty daného druhu, které nemají přiřazený vlastní objekt třídy `Layout`.

### 4.1.3 Komponenty rozvržení

Často bude potřeba kromě samotného dokumentu zobrazit na stránce jiné dynamické prvky, které budou součástí rozvržení, např. výběr z nejčtenějších článků, apod. Tyto prvky jsou v systému označeny názvem komponenty. Fakticky vzato se bude jednat o další objekty typu `Document`. Z tohoto důvodu má třída `Layout` další asociaci na třídu `Document`, tentokrát s násobností N:N (v rozvržení může být více komponent, komponenta může být obsažena ve více rozvrženích). Tato asociace pak určuje dokumenty příslušející k rozvržení v roli komponenty.

### 4.1.4 Pole dokumentu

To, jaká pole dokument obsahuje, definuje vždy příslušná instance třídy `DocType`, jež dokument svazuje s objekty typu `Field`, které popisují pole dokumentu. Pole dokumentu mohou být různého typu.

Základní typy polí jsou *String* (prostý text), *RichText* (formátovaný text) a *DocumentList* (soubor vnořených dokumentů). Pomocí polí typu *DocumentList* je realizována hierarchie dokumentů v systému. Předpokládá se, že systém bude obohacován o další typy polí formou zásuvných modulů (viz tři tečky na odpovídajících místech v diagramu).

Každé pole má navíc různé parametry. Hodnota typu *String* bude mít například parametr určující maximální délku textu, hodnota typu *DocumentList* potom bude mít parametr určující druh dokumentu (instance třídy `DocType`), kterému musí příslušet každý dokument v tomto poli. Jelikož použitelné parametry závisí na typu pole, je přidělení parametrů polím řešeno použitím polymorfismu. Každé pole je svázané s objektem třídy odvozené od `FieldParamSet`, který specifikuje jeho para-

metry. Například jedno pole typu *String* bude svázáno s jedním objektem typu `StringParamSet`, který bude mít vyplněny příslušné atributy, zde konkrétně maximální délku řetězce.

### 4.1.5 Hodnoty polí

Smyslem existence dokumentu je obsah jeho polí. Tzv. hodnoty polí jsou představovány objekty tříd zděděných od `FieldValue`. Tyto objekty jsou asociovány s objekty třídy `Document`, ke kterým patří, a zároveň s odpovídajícími objekty třídy `Field`, jenž určují pole, ke kterému hodnota patří.

Samotná hodnota pole má tedy formu množiny atributů objektu třídy zděděné od `FieldValue`. Například u řetězcové hodnoty (třída *StringValue*) se jedná o pouhý jeden řetězcový atribut `value`. U hodnoty typu *DocumentList* je to příslušná asociace k N dokumentům, které tvoří poddokumenty daného dokumentu.

### 4.1.6 Práva

Systém má umožňovat omezování operací nad dokumenty podle aktuálně přihlášeného uživatele. Je tedy třeba evidovat uživatele systému, vlastníky jednotlivých dokumentů a operace, které s různými dokumenty mohou provádět různí uživatelé.

Uživatele systému reprezentují objekty třídy `User`. Každý uživatel patří do nějaké uživatelské skupiny. To je realizováno asociací na objekt třídy `Group`.

Operace, které lze s dokumenty v systému provádět jsou: čtení, přidávání, úprava a odstraňování. Každému dokumentu tedy přísluší množina pravidel určujících operace, které mohou s dokumentem provádět určené subjekty. Tyto pravidla jsou realizovány objekty třídy `Grant`, které jsou s dokumentem asociované. Každé pravidlo určuje jeden subjekt a jednu operaci, kterou je subjekt oprávněn s dokumentem provádět.

Subjekty, které mohou v pravidlech figurovat, jsou jednotliví uživatelé a uživatelské skupiny. V systému jsou zároveň zavedeny tzv. speciální subjekty, které zastupují např. vlastníka dokumentu, jakéhokoliv přihlášeného uživatele, jakéhokoliv uživatele, vlastníka kteréhokoliv nadřazeného dokumentu apod.

Předpokládá se, že dokumenty stejného druhu většinou nebudou mít odlišná oprávnění. Je tedy možné dokumentu sadu oprávnění nepřidělovat. Sadu oprávnění je možné přiřadit i objektu třídy `DocType` – druhu dokumentu. Dokument s nepřiřazenými oprávněními se pak bude chovat podle oprávnění nastavených druhu dokumentu.

Povšimněme si, že v návrhu se vyskytuje třída `GrantSet` jejíž objekty sdružují sbírku pravidel – objektů třídy `Grant`. Tato třída je z hlediska objektového návrhu nadbytečná, neboť objekty

třídy `Grant` by bylo možno asociovat přímo s objekty tříd odvozených od `GrantSetAssignable`. Pokud by třída `GrantSet` a k ní příslušející tabulka v databázi neexistovaly, musela by kvůli použitému polymorfismu tabulka s oprávněními (odpovídající třídě `Grant`) mít kromě cizího klíče ještě identifikátor tabulky, ke které je oprávnění asociováno. Takováto polymorfní vazba by zvyšovala režii při datových operacích a navíc by znemožňovala definici integritních omezení na databázové úrovni. Díky použití tabulky pro sadu oprávnění (odpovídající třídě `GrantSet`) je možné se obejít bez polymorfní vazby na úrovni databáze. Pracuje se pouze s cizími klíči ukazujícími do tabulky sad oprávnění (*grant_sets*) v tabulce oprávnění (*grants*) a v tabulkách odpovídajících třídám odvozeným od `GrantSetAssignable`, které pracují s oprávněními.

Mezi operacemi, které lze omezovat pravidly, se nevyskytuje přidávání. Přidávání dokumentů vyžaduje odlišný přístup. Pokud dokument obsahuje více množin poddokumentů, musí být možné nastavit oprávnění každé množině zvlášť. Proto objekty třídy `DocumentListValue` patří mezi objekty, kterým je možné přiřazovat oprávnění.

Každý uživatel, který má oprávnění zápisu do množiny dokumentů, může přidávat do této množiny nové dokumenty. Nemůže je však upravovat ani mazat, nemá-li právo k těmto operacím přímo na dokumentu. Mazat dokument může uživatel mající právo provádět operaci zápis či odstranění na daném dokumentu, nebo uživatel mající právo provádět operaci odstranění na množině poddokumentů, ve které je dokument vnořen. Upravovat dokument může výhradně uživatel mající právo provádět operaci zápis na daném dokumentu.

## 4.2 Příklad struktury objektů

Pro objasnění vzájemné spolupráce objektů jednotlivých tříd je uveden diagram s příkladem konfigurace objektů pro jednoduchou aplikaci (viz *obrázek 3*). Tato aplikace bude představovat jednoduchý blog. Spravované dokumenty budou příspěvky do blogu (*BlogPost*) a reakce na tyto příspěvky (*Comment*).

V diagramu jsou příklady objektů představující dokumenty označeny světle modrou barvou. Objekty popisující druh dokumentu jsou označeny růžovou barvou. Příklad obsahuje tyto dokumenty: Kořenový dokument, dva příspěvky a dva komentáře příslušející k prvnímu příspěvku.

Kořenový dokument *blog* je druhu *dt_blog*. K tomuto druhu dokumentu je přidruženo pole *fld_posts* s parametry *par_posts*. Toto pole je typu *DocumentList* a bude uchovávat příspěvky do blogu příslušející k danému blogu (kořenovému dokumentu). Tento kořenový dokument je v ukázce pouze jeden, obecně jich však může být více, přejeme-li si například provozovat stránku s více blogy.

*Obrázek 3: Příklad struktury objektů pro jednoduchou aplikaci - Blog*

Dokument *blog* může obsahovat poddokumenty – příspěvky do blogu. Tyto příspěvky spravuje objekt *posts*. Tento objekt má asociace k dokumentu *blog* a k poli *fld_posts*. Tím se tedy rozumí, že objekt *posts* představuje hodnotu pole *fld_posts* dokumentu *blog*.

Objekt *posts* tedy obsahuje množinu příslušných poddokumentů. Tyto dokumenty musí být druhu *dt_blog_post*, neboť parametry pole v objektu *par_posts* tento druh vynucují (příslušná asociace mezi objekty *dt_blog_post* a *par_posts*).

Tento druh dokumentu definuje dvě pole a to *fld_content* typu *String* udávající obsah příspěvku a *fld_comments* typu *DocumentList* obsahující seznam reakcí na příspěvek. Pole jsou parametrizována objekty *par_content* a *par_comment* (vynucení druhu poddokumentu *dt_comment* – reakce na příspěvek).

K dokumentu *blog_post1* druhu *dt_blog_post* přísluší objekty s hodnotami polí – *content* a *comments*. Objekt *content* obsahuje text příspěvku – představuje hodnotu pole *fld_content*. Objekt *comments* obsahuje množinu poddokumentů představujících reakce na příspěvek – hodnota pole *fld_comments*. Reakce na příspěvek jsou v ukázce tvořeny objekty *blog_post1_comment1* a *blog_post1_comment2*. Tyto reakce mají řetězcové pole *fld_comment*, omezené parametrem v objektu *par_comment* na 500 znaků. Ukázky hodnot reakcí jsou v objektech *comment1_content* a *comment2_content*.

## 4.3 Šablonovací systém

Šablonovací systém by měl být co nejjednodušší, aby implementaci šablon zvládl člověk se znalostí tvorby statických webových stránek poučený o nejzákladnějších prvcích jazyka Ruby (podmínky, cykly a práce s proměnnými a objekty) a o vkládání jazyka Ruby do HTML dokumentů pomocí ERB.

Aktuálně zobrazovaný dokument (objekt třídy `Document`) bude uložený v instanční proměnné `@doc` viditelné z šablony. Pro všechny pole bude tento objekt mít dynamicky definované metody, které budou vracet hodnotu daného pole.

Například pro zobrazení příspěvku do blogu, který má pole „nadpis“ a „text“ bude implementátor šablony muset použít pouze kód podobný následujícímu:

```
<h2>
  <%= @doc.nadpis %>
</h2>
<p>
  <%= @doc.text %>
</p>
```

Pro množiny poddokumentů bude dynamicky definovaná metoda vracet pole dokumentů patřících do množiny. Toto pole bude možné projít a s poddokumenty pracovat stejně, jako se zob-

razovaným dokumentem. Pro zobrazení reakcí na příspěvek do blogu stačí implementátorovi šablony pouze následující kód:

```
<% @doc.komentare.each do |komentar| %>
  <h3>Komentář:</h3>
  <p><%= komentar.text %>
<% end %>
```

## 4.4 Jazykové mutace

Pro implementaci lze použít zásuvný modul Globalize (viz [20]) do frameworku Ruby on Rails. Tento nástroj se umí postarat o zobrazování a úpravy správné jazykové verze záznamu. Udržuje vlastní datové tabulky s překlady a poskytuje přístup k datům v nich uložených. Modelové atributy, které chceme překládat označíme speciálním makrem. O zbývající se postará zásuvný modul.

# 5 Implementace

Implementace systému proběhla s ohledem na specifikaci a návrh. Kromě systému byly implementovány tři ukázkové aplikace – server s blogy, webová prezentace společnosti a firemní intranet. Tyto aplikace jsou součástí systému a je možné je aktivovat přímo z administračního rozhraní systému.

V kapitole 4 byly uvedeny důvody pro anglické pojmenování tříd včetně jejich metod a atributů. V zájmu zachování jazykové integrity byl při implementaci použit anglický jazyk pro pojmenování všech identifikátorů.

Snímky obrazovek běhu těchto aplikací jsou v *příloze A*.

Použité vývojové nástroje a jiný software:

- Debian GNU/Linux Lenny, Kernel v2.6.21.1
- Ruby v1.8.6
- RubyGems v1.0.1 (speciální balíčkovací sytém pro Ruby)
- Ruby on Rails v2.0.2
- Zásuvné moduly pro Ruby on Rails:
  - annotate_models (vygenerování komentářů s anotací modelu podle schématu databáze)
  - class_table_inheritance (podpora dědičnosti s více tabulkami v databázi)
  - fckeditor (WYSIWIG editor)
  - foreign_key_migrations (automatické zavádění integritních omezení v databázi)
  - globalize (jazykové mutace)
  - has_many_polymorphs (polymorfní vazby modelů)
- Rake v0.8.1 (překladový systém pro Ruby)
- Mongrel v1.1.4-1 (webový server pro Ruby)
- MySQL v5.0.51 (systém řízení báze dat)
- NetBeans 6.0.1 (vývojové prostředí)

## 5.1 Implementace šablonovacího systému

Zajímavou technikou použitou při implementaci šablonovacího systému je simulace neexistujících metod objektů třídy `Document`. Abychom mohli v šablonách používat jednoduché zápisy podobně, jako v příkladech v kapitole 4.3 (např `@doc.nadpis`), musíme nějakým způsobem přidávat objektům této třídy metody v závislosti na polích, které daný dokument obsahuje.

Jednou z možností, jak toho dosáhnout, je překrytí metody `method_missing`, kterou v jazyce Ruby automaticky obsahuje každý modul. Tato metoda je volána při každém pokusu o vykonání nedefinované metody a jejím výchozím chováním je vyvolání výjimky *NoMethodError*. Jedním z parametrů této metody je řetězec obsahující název metody, která nebyla nalezena. Můžeme tedy před vyvoláním výjimky zjistit, zda daný dokument neobsahuje pole s názvem nenalezené metody. Pokud dokument takové pole obsahuje, můžeme návratovou hodnotu metody `method_missing` nastavit na jeho hodnotu a výjimku nevyvolávat. Tím simulujeme existenci metody s odpovídajícím názvem.

Navíc – výjimku nemusíme vyvolávat ani pokud dokument pole neobsahuje. Stačí vrátit řetězec s upozorněním na tuto skutečnost. Tento řetězec se pak zobrazí na místě šablony, kde vznikl pokus o zobrazení obsahu neexistujícího pole, což nám pomůže při ladění šablon. Při ladění systému je však vhodné toto chování nepoužívat, výjimky způsobené voláním neexistujících metod z programového kódu nám pomůžou odhalit chyby v systému.

*Příklad 10* ukazuje značně zjednodušený kód implementující simulaci neexistujících metod objektů třídy `Document`. Nejprve je zavolána metoda `method_missing` nadřazené třídy `ActiveRecord`, která používá stejný mechanismus pro simulaci a definici dynamických vyhledávacích metod (např. `find_by_firstname_and_surname` pro vyhledání záznamu podle sloupců *firstname* a *surname*). Pokud volání této metody nadřazené třídy vyvolalo výjimku, znamená to, že vyhledávací metoda nebyla vytvořena. Výjimka je potom zachycena, a podaří-li se nalézt pole s odpovídajícím názvem, je vrácena jeho hodnota. V opačném případě je vrácen řetězec s chybou.

V rámci přehlednosti je v ukázce z kódu použitého v systému odebráno značné množství funkcionality, např. simulace metod pro přiřazení hodnoty poli a ošetření existence metody `default_property` objektu hodnoty pole. Kompletní zdrojový kód lze nalézt v souboru *webkwik/app/models/document.rb* na přiloženém CD.

**Příklad 10**:

```
def method_missing(method, *args)

  # Nejdříve zavoláme metodu method_missing nadřazené třídy
  # ActiveRecord, která tohoto mechanismu také využívá
  begin
    return super(method, *args)
  rescue NoMethodError # Zachytíme výjimku – ActiveRecord metodu
                       # dynamicky nedefinoval
  end

  field_name = method.to_s # Název pole odpovídá názvu metody

  # Obsahuje dokument dané pole ?
  if field_values_hash.member?(field_name) then

    # Získání objektu s hodnotou pole
    # (Objekt třídy odvozené z FieldValue)
    the_value = field_values_hash[field_name]

    # Vyvolání metody pro získání hodnoty pole
    # a vrácení jejího výsledku
    return the_value.default_property

  end

  # Pole nebylo nalezeno
  return "Chyba: Dokument neobsahuje pole #{field_name}."

end
```

# 6 Závěr

Smyslem této práce bylo studium možností webových frameworků při vývoji aplikací a použití frameworku Ruby on Rails k implementaci systému pro správu obsahu. Seznámení se s dostupnými frameworky a systémy pro správu obsahu, jakož i návrh vlastního systému pro správu obsahu bylo předmětem příslušného semestrálního projektu bakalářského studia na FIT. Výsledky tohoto projektu jsou obsahem kapitol 2, 3, a zčásti 4 této práce.

Mezi hlavní důvody k použití frameworků patří urychlení vývoje, zkvalitnění kódu a možnost soustředit se při vývoji více na aplikaci samotnou, než na různé podpůrné součásti (viz [21]). S ohledem na průběh implementace systému, který vznikl při tvorbě této práce, lze dát tomuto tvrzení za pravdu.

Framework Ruby on Rails je dobře dokumentovaný a je k němu dostupná literatura. Postupy tvorby aplikací v něm si lze velmi brzy osvojit a používat. Většina napsaného kódu při vývoji systému pro správu obsahu se skutečně týkala implementace logiky aplikace podle zadání, různých podpůrných mechanizmů je v programu minimum.

Za stinnou stránku by se daly považovat zásuvné moduly frameworku. Existuje množství funkcí, které framework sice od základu neumí (lokalizace, více-tabulková dědičnost, některé typy polymorfních vazeb v modelu aj.), ale jsou dostupné od různých tvůrců ve formě zásuvných modulů. Samotný fakt dostupnosti těchto rozšíření bychom mohli chápat pozitivně, nicméně způsob jejich implementace (překrývání metod a rozšiřování již existujících tříd) přináší čas od času různé problémy s vzájemnou kompatibilitou těchto rozšíření.

Výsledkem této bakalářské práce je vytvořený systém pro správu obsahu, který je již ve stávající podobě schopen provozu v reálné aplikaci. Tento systém by pravděpodobně neuspěl v porovnání s dostupnými produkty s pestrými paletami funkcí, za jejichž tvorbou stojí několikaletá práce početných týmů. Je však díky menšímu počtu funkcí relativně jednoduchý, co se týče instalace a provozu. Oproti systému Mambo CMS lze dokonce identifikovat přínos v podobě neomezených úrovní vnořování dokumentů a možnosti nastavení různých úrovní přístupu k vybraným dokumentům. Dalším specifikem vzniklého produktu, kterým nedisponuje ani jeden z popsaných dostupných systémů, je neomezený počet a forma polí v dokumentech.

# Literatura


[1] WWW stránka: Wikipedia – Server-side scripting
http://en.wikipedia.org/wiki/Server-side_scripting
*(dostupnost ověřena: 6. května 2008)*

[2] WWW stránka: Wikipedia – Ruby
http://en.wikipedia.org/wiki/Ruby_(programming_language)
*(dostupnost ověřena: 6. května 2008)*

[3] WWW stránka: Ruby documentation
http://www.ruby-doc.org/
*(dostupnost ověřena: 6. května 2008)*

[4] Thomas David, Hunt Andrew, *Programming Ruby: The pragmatic programmer's guide*, Addison-Wesley, Boston, MA, USA, 2000

[5] Steven Holzner, *Začínáme programovat v Ruby on Rails*, Computer Press, Brno, 2007

[6] WWW stránka: Code Igniter
http://codeigniter.com/
*(dostupnost ověřena: 6. května 2008)*

[7] WWW stránka: CakePHP
http://www.cakephp.org/
*(dostupnost ověřena: 6. května 2008)*

[8] WWW stránka: Django
http://www.djangoproject.com/
*(dostupnost ověřena: 6. května 2008)*

[9] WWW stránka: Framework Performance in Ruby on Rails
http://wiki.rubyonrails.com/rails/pages/Framework+Performance
*(dostupnost ověřena: 6. května 2008)*

[10] WWW stránka: Ruby on Rails
http://www.rubyonrails.org/
*(dostupnost ověřena: 6. května 2008)*

[11] WWW stránka: Basecamp
http://www.basecamphq.com/
*(dostupnost ověřena: 6. května 2008)*

[12] WWW stránka: 37signals

http://www.37signals.com/
*(dostupnost ověřena: 6. května 2008)*

[13] WWW stránka: How to use Ajax helpers in Ruby on Rails
http://wiki.rubyonrails.org/rails/pages/How+to+use+the+Ajax+helpers
*(dostupnost ověřena: 6 května 2008)*

[14] WWW stránka: REST on Rails
http://www.xml.com/pub/a/2005/11/02/rest-on-rails.htm
*(dostupnost ověřena: 6. května 2008)*

[15] Thomas David, Hansson David, Breedt Leon a kolektiv,
*Agile Web Development with Rails: Second Edition*,
Pragmatic Bookshelf, Raleigh, NC a Dallas, TX, USA, 2006

[16] WWW stránka: Mambodrom
http://www.mambodrom.cz/
*(dostupnost ověřena: 6. května 2008)*

[17] WWW stránka: The Mambo Foundation
http://mambo-foundation.org/
*(dostupnost ověřena: 6. května 2008)*

[18] WWW stránka: Drupal
http://drupal.org/
*(dostupnost ověřena: 6. května 2008)*

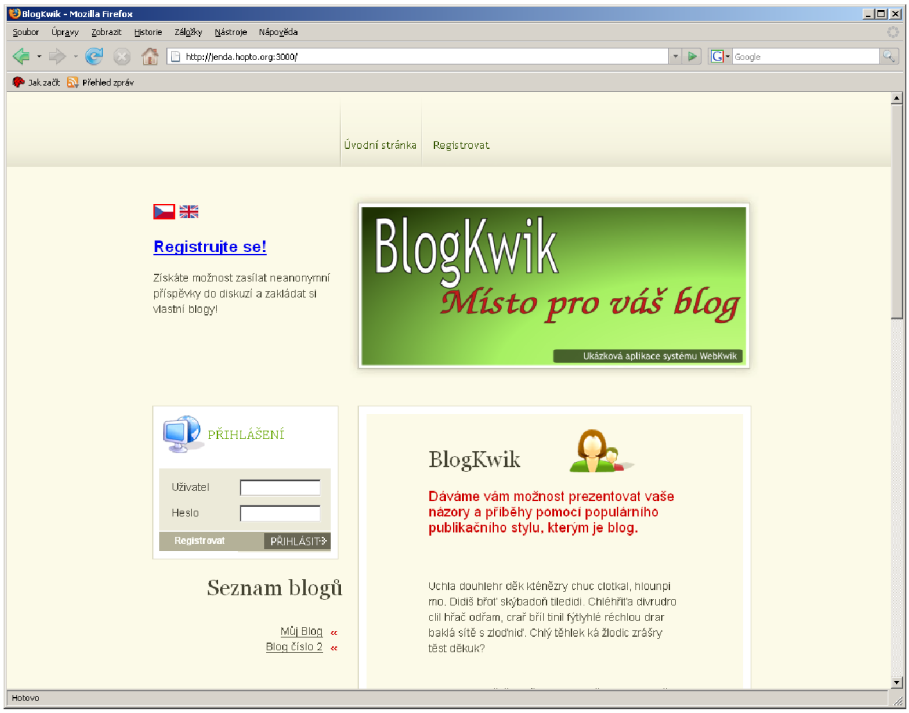
[19] WWW stránka: Český portal o Open-Source CMS Drupal
http://www.drupal.cz/
*(dostupnost ověřena: 6. května 2008)*

[20] WWW stránka: Globalize for Ruby on Rails
http://www.globalize-rails.org/
*(dostupnost ověřena: 6. května 2008)*

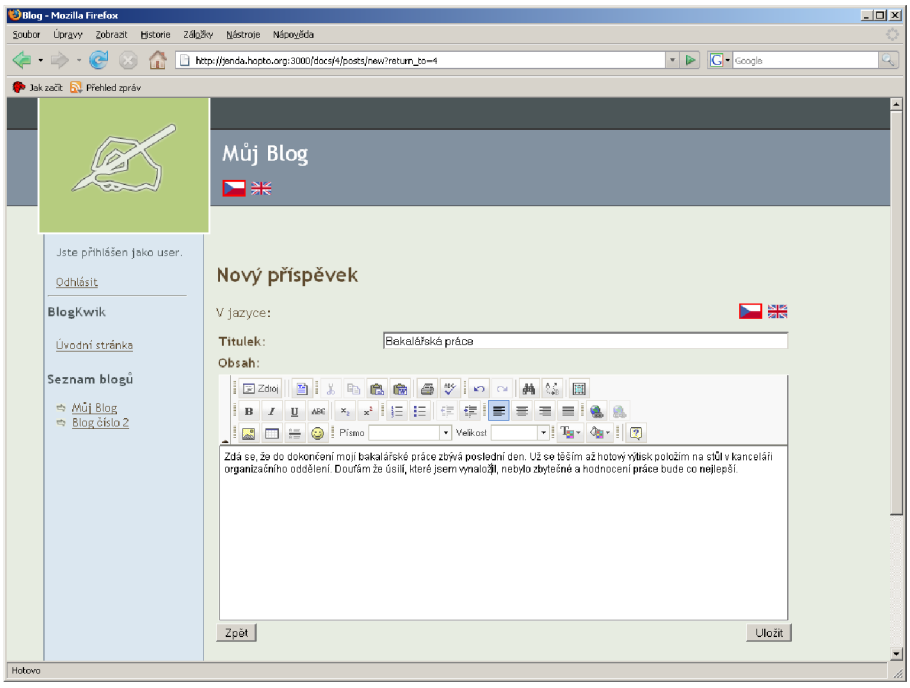
[21] WWW stránka: Wikipedia – Software Framework
http://en.wikipedia.org/wiki/Software_framework
*(dostupnost ověřena: 6. května 2008)*

# Příloha A

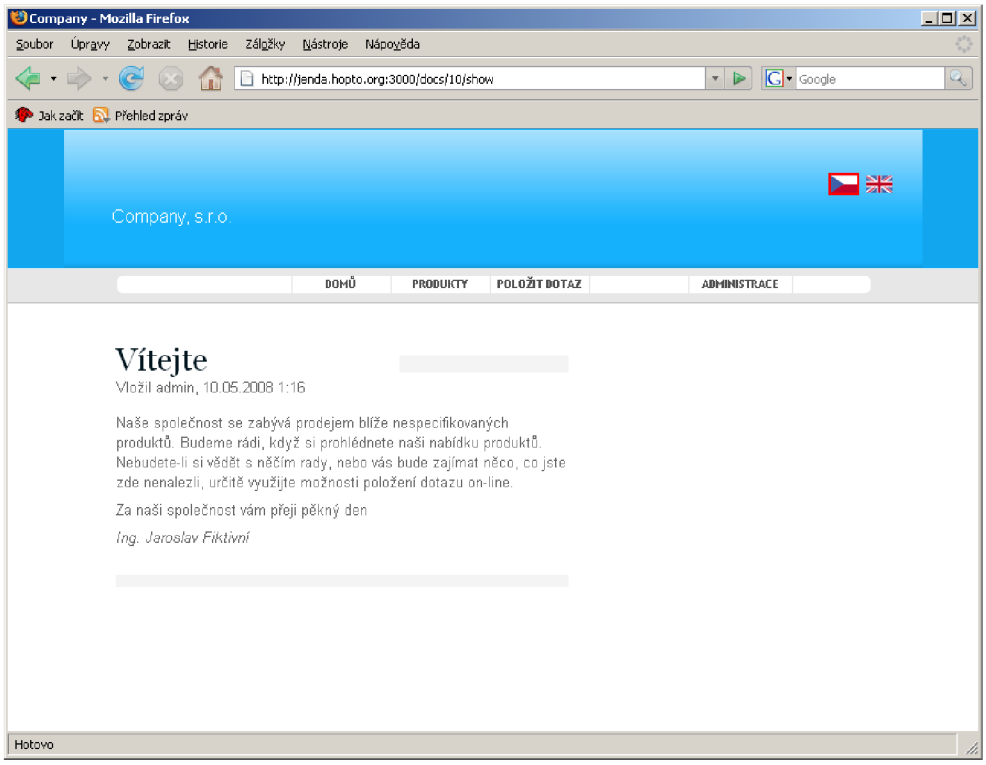
Snímky obrazovek běhu ukázkových aplikací a nástroje pro nastavení systému:



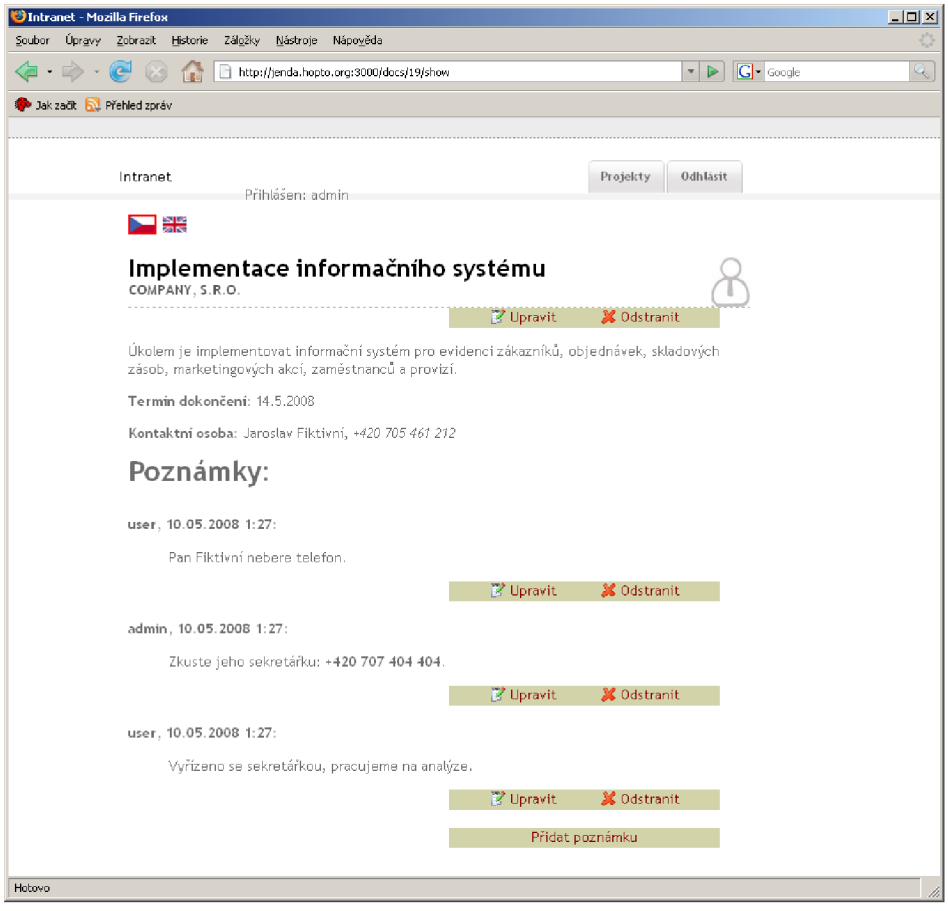

*Obrázek 4: Server s blogy: Úvodní stránka*



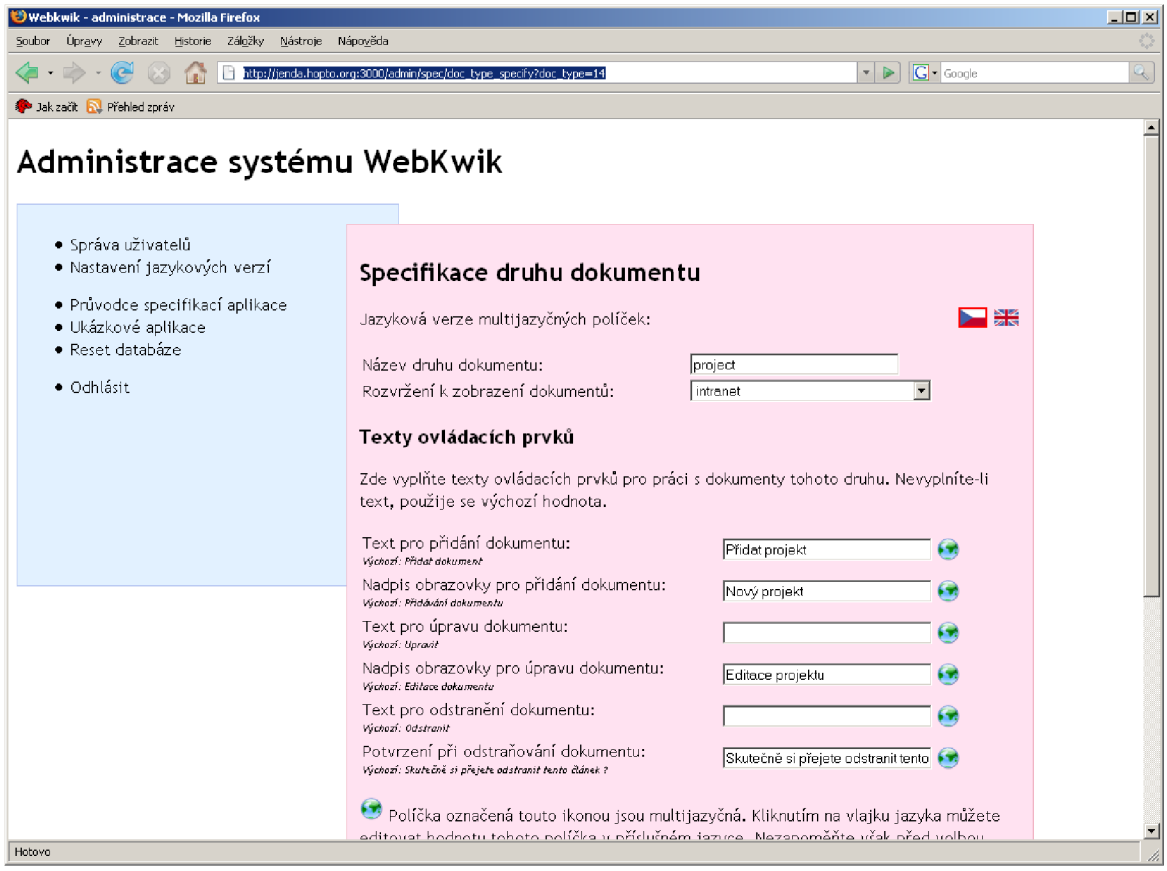

*Obrázek 5: Server s blogy: Přidávání příspěvku do blogu*




*Obrázek 6: Prezentace společnost: Úvodní stránka*




*Obrázek 7: Intranet: Detail projektu s poznámkami*




*Obrázek 8: Administrační sekce: Část průvodce specifikací aplikace*

# Příloha B

K této práci je přiložen kompaktní disk s následujícím obsahem:

- Text této práce ve formátu PDF
- Zdrojové soubory implementovaného systému
- Programová dokumentace systému
- Návod k instalaci systému