UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
Matematika - praktična matematika (VSŠ)

Andrej Kovič

Razvoj spletne aplikacije z uporabo

Ruby on Rails

Diplomska naloga

Ljubljana, 2011

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

Zahvala

Za strokovno pomoč, čas in trud vložen pri sestavljanju in popravkih diplome, bi se rad zahvalil mentorju mag. Matiji Lokarju.

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

Program diplomske naloge

V okviru diplomske naloge proučite, kako s sistemom Ruby on Rails razvijemo spletno aplikacijo. Ob tem prikažite tudi osnove jezika Ruby.

Osnovna literatura:

- D. Griffiths, Head First Rails, O'Reilly Media, 2008
- S. St. Laurent, E. Dumbill, Learning Rails, O'Reilly Media, 2008

mentor

mag. Matija Lokar

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

Povzetek

Diplomska naloga je namenjena študentom Fakultete za matematiko in fiziko, ki si želijo pridobiti nova znanja na področju spletnega programiranja v okolju Ruby on Rails.

Diplomska naloga je sestavljena iz pisnega dela in učne spletne aplikacije. Sestavljajo jo tri večja poglavja:

- *Ruby*
- *Rails*
- *Razvoj aplikacije*

V prvem poglavju diplome so predstavljene osnove programskega jezika Ruby. V poglavju Rails je predstavljeno razvojno okolje Rails, ki je napisano v jeziku Ruby. V zadnjem poglavju Razvoj aplikacije preučimo razvoj preproste učne spletne aplikacije tipa spletna knjiga.

Abstract

The suitable audience for the thesis are students of University of mathematics and physics, which want to aquire new knowledge in the area of web programming in Ruby on Rails environment.

The thesis consists of the written part and a learning web application. The written part is assembled from three major chapters:

- *Ruby*
- *Rails*
- *Application development*

The basics of the programming language Ruby are presented in the first chapter of the thesis. The chapter Rails contains a description of Rails development platform, which is written in programming language Ruby. In the last chapter Application development we study the development of a simple learning web-book application.

Math. Subj. Class. (2011): 68M12**,** 68N01, 68N15, 68N18, 68N19, 68N20

Computing Review Class. System (1998): A.1, D2.6, D.3.0, D3.1, D3.2, D.3.3

Ključne besede: spletna aplikacija, Ruby, Rails, paket Ruby, paket Rails, Python, niz, tabela, pogojni stavek, zanka, slovar, razred, modul, metoda, blok, procedura, simbol, izjema, generator, spletni strežnik, imenik, datoteka, model, krmilnik, prikazovalnik, usmeritev, spletni naslov, baza podatkov, akcija, vzorčna datoteka

Keywords: web application, Ruby, Rails, gem, Railtie, Python, string, table, conditional, loop, doctionary, class, module, method, block, procedure, symbol, exception, generator, web server, directory, file, model, controller, view, route, web address, database, action, template file

Kazalo

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

1 Uvod

Diplomska naloga opisuje razvijanje spletne aplikacije v okolju Ruby on Rails. Njen cilj je predstaviti jezik Ruby in okolje Rails. Razdeljena je na tri večja poglavja. V prvem poglavju bralec spozna osnove jezika Ruby. Osnovne značilnosti jezika so predstavljene s primeri, ki so podkrepljeni s primeri enakega pomena v jeziku Python. V tem poglavju naj bi bralec osvojil osnovno znanje jezika Ruby, ki ga bo potreboval za razumevanje drugih dveh poglavij. V drugem poglavju z naslovom Rails spoznamo delovanje spletnih aplikacij v okolju Rails. Povemo, kako je sestavljena spletna aplikacija in kako deluje. V zadnjem poglavju Razvoj aplikacije se posvetimo podrobnostim pri implementaciji preproste spletne aplikacije v okolju Rails. Razvili bomo spletno knjigo, v katero bomo shranjevali razdelke te diplomske naloge z vsemi podatki in slikami. Pri tem bomo uporabili znanje, ki smo ga pridobili v prvih dveh poglavjih.
Diplomska naloga bi morala bralcu pomagati pri osvajanju osnovnega znanja jezika Ruby in spoznavanju delovanja spletnih aplikacij v okolju Rails. Avtor Andrej Kovič sem si želel predvsem pridobiti nova znanja na področju spletnega programiranja. Ta diplomska naloga v svojem bistvu opisuje proces učenja jezika Ruby in spoznavanja njegove uporabe v okolju Rails. Priložena je tudi vsa koda spletne aplikacije, ki sem jo uporabil kot zgled.

2 Ruby

2.1 Zgodovina

Avtor programskega jezika Ruby je Yukihiro Matsumoto - Matz. Prva različica jezika Ruby je bila zasnovana v začetku leta 1993. Avtor je kot motiv za nastanek novega jezika navedel, da si je želel ustvariti jezik, ki bi bil bolj raznovrsten kot Perl in bolj objektno orientiran kot Python. Prevajalnik jezika Ruby je napisan v programskem jeziku C. Prva javno objavljena različica jezika je bila Ruby 0.95. Objavljena je bila le lokalno na Japonskem 21. decembra 1995. Naslednja različica jezika Ruby 1.0 je bila izdana 25. decembra leta 1996. Ta različica je bila popolnoma objektno orientirana. Omogočila je uporabo principa dedovanja pri sestavljanju razredov, vključevanje modulov v razrede, uporabo iteratorjev pri delu z seznami objektov, in avtomatično čiščenje pomnilnika po zaključenem življenskem ciklu objektov. Šele leta 1999 pa je različica Ruby 1.3 pritegnila programerje izven Japonske. Informacije, povezane s programiranjem v jeziku Ruby, so si programerji začeli izmenjevati v obliki sporočil na posebnem spletnem portalu, imenovanem Ruby-Talk. S tem se je uporaba Rubyja začela širiti po svetu. V letu 2000 je bila izdana prva knjiga v angleškem jeziku z naslovom "Programming Ruby". Pravi razcvet pa je Ruby doživel leta 2005 z nastankom okolja Rails. Gre za sistem knjižnic razredov, ki je omogočil hiter in enostaven razvoj spletnih aplikacij. Prav pojav tega okolja je pripomogel k hitri širitvi uporabe jezika Ruby.
Zadnja (september 2011) izdana različica jezika Ruby je 1.9.2. Pri razvoju spletne aplikacije, ki jo predstavljamo v diplomski nalogi, smo uporabili različico 1.8.7.
V letu 2010 so se pojavile alternativne različice jezika Ruby:

- YARV – (Yet another Ruby) je nova različica ukazne vrstice, razvite za uporabo v okolju programskega jezika Ruby. Napisana je, da bi se izboljšala hitrost izvajanja programov, pisanih v jeziku Ruby. Od različice 1.9 jezika Ruby YARV uporabljamo za izvajanje ukazov v ukazni vrstici.
- JRuby – različica okolja programskega jezika Ruby, napisana v Javi.
- Rubinius – različica okolja programskega jezika Ruby, ki je zasnovana na principih jezika SmallTalk.

- IronRuby – različica okolja programskega jezika Ruby, zasnovana za delo z okoljem Microsoft .NET.
- MacRuby – različica okolja programskega jezika Ruby, napisana v jeziku Objective-C z uporabo razvojnega okolja CoreFoundation. Razvojno okolje CoreFoundation se uporablja za razvijanje aplikacij na operacijskem sistemu Apple OSX.

Kot vidimo, so različice jezika Ruby namenjene izboljšavam in prilagoditvam jezika Ruby na različne razvijalske platforme in operacijske sisteme.
Osnovna različica jezika Ruby povzema določene lastnosti jezikov Perl, Python, SmallTalk, Eiffel, Ada in Lisp. Na ta način omogoča tako uporabo funkcijskega načina programiranja, kot tudi objektni način programiranja.

2.2 Lastnosti

V tem razdelku si bomo ogledali osnovne značilnosti jezika Ruby.
Ruby je objektno orientiran jezik. Vse v Rubyju je objekt. Deklaracija spremenljivk ni potrebna, tip objekta se dinamično določi pri inicializaciji. Kot bomo videli v naslednjih razdelkih, vsi razredi posredno ali neposredno dedujejo iz osnovnega razreda `Object`. Zelo uporabna lastnost jezika Ruby je tudi dejstvo, da lahko vsem razredom dodajamo metode. To pomeni, da tudi standardni razredi, kot je na primer razred `Object` , niso zaklenjeni. Če pri razvoju programa v jeziku Ruby ugotovimo, da nam v določenem razredu manjka metoda, ta razred preprosto nadgradimo brez da bi z uporabo dedovanja izpeljali nov razred tako, da enostavno obstoječi razred dopolnimo z definicijo nove metode.
Ruby je tudi skriptni jezik. V datoteke s končnico `.rb` shranjujemo zaporedne ukaze. Skripto podamo kot parameter ukazu `/usr/bin/ruby`, ki zaporedoma izvede ukaze, napisane v skripti. V standardnem okolju programskega jezika Ruby obstaja tudi ukazna vrstica, imenovana `irb`. Z ukazom `irb` pokličemo ukazno vrstico, v kateri lahko izvajamo poljubne ukaze jezika Ruby.
Ruby omogoča, da med delovanjem programa dinamično pridobimo in uporabimo informacije o razredu, ki mu pripada posamezen objekt. To značilnost jezika Ruby imenujemo `Reflection`.

Oglejmo si sedaj še nekaj splošnih pravil pri pisanju kode v jeziku Ruby. Lastnosti jezika bomo predstavili s preprostimi primeri uporabe.
Najprej bomo v jeziku Ruby napisali preprosto skripto tipa "`Hello World`". V datoteko z imenom `leppozdrav.rb` shranimo naslednjo kodo:

```
#!/usr/bin/ruby

puts "Lep pozdrav!"
```

Z vrstico `#!/usr/bin/env ruby` smo določili, da je vsebina naše datoteke skripta v jeziku Ruby. Torej se bo pri izvajanju skripte pognal ukaz `/usr/bin/ruby`, skripta pa mu bo podana kot parameter. Prvi (in edini) ukaz te skripte je ukaz `puts`, s katerim na zaslon izpišemo niz znakov. Ukaz `puts` je del modula `Kernel`, ki je v osnovnem razredu `Object` avtomatsko vključen. Skripto izvedemo z ukazom:

```
andrej@andrej-desktop:~/ruby$ ./leppozdrav.rb
```

Odziv sistema je seveda:

```
Lep pozdrav!
```

V naslednjem zgledu si bomo ogledali, kako poženemo ukazno vrstico `irb`. Z ukazom

```
andrej@andrej-desktop:~/ruby$ irb

>
```

poženemo ukazno vrstico `irb`. Znak > je pozivnik ukazne vrstice `irb`. Ime `irb` (`Interactive Ruby`) izhaja iz lastnosti, da se vsak vnešen ukaz izvede ob pritisku na tipko `Enter`. Izračunana vrednost vnešenega ukaza se izpiše v zadnji vrstici izpisa v oknu terminala. Ob izvedbi posameznega ukaza bomo torej na koncu izpisa vedno dobili še rezultat ukaza. Ta rezultat je

lahko poljuben objekt kateregakoli razreda jezika Ruby.
V primerih izvajanja ukazov z `irb`, ki sledijo, bomo pogosto opazili, da se je ob koncu izvajanja ukaza izpisala vrstica "`=> nil`". V takih primerih je izvedeni ukaz vrnil vrednost `nil`. Z `nil` označimo ničelni objekt, ki predstavlja rezerviran naslov v pomnilniku z vrednostjo `0x00000000`.
Ukazno vrstico bomo uporabili za prikaz poglavitnih lastnosti jezika.
Najprej si oglejmo preprost primer, s katerim bomo prikazali, da je prav vse v Rubyju objekt. V nasprotju z jezikom Python, kjer je npr. 5 število (in ni objekt), je v Rubyu tudi 5 dejansko objekt. Prav tako vidimo, da je parameter ukaza lahko tudi akcija – v našem primeru klic ukaza za izpisovanje:

```
> 5.times { print " Vse v Rubyju je objekt\n" }
 Vse v Rubyju je objekt
 Vse v Rubyju je objekt
 Vse v Rubyju je objekt
 Vse v Rubyju je objekt
 Vse v Rubyju je objekt
=> 5
```

Vsak razred v Rubyju deduje iz razreda `Object`. V uvodu smo omenili dinamično pridobivanje podatkov o razredih posameznih objektov. Z ukazom

```
> 5.class
=> Fixnum
```

ugotovimo, da je število 5 objekt razreda `Fixnum`. Z ukazom

```
> 5.class.ancestors - 5.class.included_modules
=> [Fixnum, Integer, Numeric, Object]
```

smo dobili verigo dedovanja razreda `Fixnum`. Metoda `class`, ki jo izvedemo nad objektom `5`, nam vrne razred `Fixnum`. Na tem razredu z metodo `ancestors` dobimo tabelo vseh prednikov in v razred `Fixnum` vključenih modulov. Slednje bomo spoznali v poglavju *Ruby, Razredi, metode in moduli*. V našem primeru jih preprosto odštejemo iz tabele rezultatov. Tako potem dobimo le prednike razreda `Fixnum`.
Jezik Ruby omogoča dedovanje le iz enega razreda. Razred `Fixnum` torej deduje iz razreda `Integer`, le ta deduje iz razreda `Numeric`, razred `Numeric` pa deduje iz osnovnega razreda `Object`.
Na primeru si oglejmo še dodajanje metod obstoječim razredom v jeziku Ruby. V Rubyju namreč razredi niso zaklenjeni. Zato jih lahko brez uporabe dedovanja preprosto nadgrajujemo. Če potrebujemo določeno drugačno metodo seštevanja števil, lahko v razred `Numeric` dodamo metodo `sestejInPovecaj`. V našem primeru želimo, da nam metoda vrne vsoto števil + 1.

```
> class Numeric
>    def sestejInPovecaj(num)
>      self.+(num+1)
>    end
> end
=> nil
> 5.sestejInPovecaj 6
=> 12
```


Na ta način lahko poljubnemu razredu preprosto dodamo metode, ki jih potrebujemo.

Sedaj bomo predstavili še nekaj splošnih pravil, ki jih moramo upoštevati pri pisanju kode v jeziku Ruby.
Pri Rubyju število presledkov in zamikanje ne igra nobene vloge. Seveda moramo pri pisanju zaradi preglednosti uporabljati zamike, vendar za samo interpretacijo kode niso potrebni.
Razredi in moduli v jeziku Ruby se vedno začnejo z veliko začetnico. Primer definicije razreda je denimo:

```
class Numeric

...

end
```

Pri klicanju metod objektov posameznih razredov ni potrebno uporabljati okroglih oklepajev. Tako sta zapisa

```
objekt.metoda( param1, param2)
```

in

```
objekt.metoda param1, param2
```

povsem ekvivalentna.
Konec modula, razreda, metode, pogojnega ukaza ali bloka ukazov v kodi vedno zaključimo s ključno besedico `end`. Vsaka metoda razreda jezika Ruby avtomatsko vrača kot vrednost metode zadnjo v metodi izračunano vrednost. Zato nam v metodi `sestejInPovecaj` ni bilo potrebno poklicati `return self.+(num+1)`. Zadostoval je stavek `self.+(num+1)`.
V razdelku *Ruby, Bloki in procedure* bomo spoznali še eno značilnost jezika Ruby. V tem razdelku jo omenjamo, ker je tudi s stališča pisanja kode nekoliko posebna. V vrstici primera tega razdelka

```
5.times { print " Vse v Rubyju je objekt\n" }
```

smo namreč videli primer uporabe bloka ukazov. Bloke ukazov lahko definiramo z uporabo zavitih oklepajev. Blok ukazov je vedno povezan z metodo, ki izvede ukaze napisane v njem. V našem primeru je to metoda `times` razreda `Fixnum`.
Ukaz `print "Vse v Rubyju je objekt\n"` povzroči izpis stavka, omejenega z narekovaji na standardni izhod. Izpis na standarnem izhodu

```
Vse v Rubyju je objekt
```

je v našem primeru rezultat bloka ukazov (v našem primeru vsebuje le en ukaz). Metoda `times` izvede ukaze v bloku ukazov v skladu z vrednostjo objekta razreda `Fixnum`, na katerem smo jo poklicali (v našem primeru je vrednost objekta razreda `Fixnum` enaka 5).
Torej zgornji ukaz dejansko pomeni 5-kratni izpis stavka v narekovajih na standardni izhod.

S podobnimi in še nekoliko bolj zapletenimi konstrukti se bomo srečevali v naslednjih razdelkih.

2.3 Paketi (RubyGems)

`RubyGems` je orodje za nameščanje paketov, napisanih v jeziku Ruby. Rails je primer takega paketa jezika Ruby. V tem razdelku bomo opisali različne tipe paketov. Začeli bomo s kratkim opisom pojmov, ki jih bomo uporabljali v vseh naslednjih razdelkih. Nadaljevali bomo s primerom nameščanja in pregledovanja nameščenih paketov z uporabo orodja `RubyGems`.
Najprej bomo spoznali ključne besede, s katerimi bomo v naslednjih poglavjih poimenovali različne tipe paketov okolja Rails.

Aplikacija v jeziku Ruby je datoteka ali skupek datotek s končnico `.rb`, v katerih je napisan program v jeziku Ruby. Datoteko z ukazi podamo kot parameter izvršilni datoteki `/usr/bin/ruby`, ki te ukaze izvrši. Aplikacije, ki jih sestavlja ena sama datoteka s končnico `.rb`, bomo v naslednjih razdelkih pogosto poimenovali tudi **skripta**.
Knjižnica v jeziku Ruby je skupek datotek, napisanih v jeziku Ruby. Knjižnica vsebuje funkcije,

ki jih lahko uporabimo v aplikaciji, potem ko knjižnico naložimo v spomin objekta tekoče aplikacije.
Paket jezika Ruby (`gem`) vsebuje aplikacijo ali knjižnico, napisano v jeziku Ruby. Da bi določeno aplikacijo ali knjižnico lažje namestili na sistem, mora biti zapakirana v tak paket. Paket naredimo z orodjem `RubyGems`. Slednji nam omogoča, da ustrezno shranimo vse podatke o paketu, ki jih bomo potrebovali na sistemu. Med take podatke spada podatek o različici aplikacije (ali knjižnice), vsebovane v paketu, namestitvena pot aplikacije in informacije o paketih, od katerih je aplikacija odvisna.
Različica aplikacije je pomembna, ker z njo laže nadgradimo aplikacijo na novejšo verzijo. Informacija o sistemski poti aplikacije se prav tako uporablja pri nameščanju novejše verzije ali pri odstranjevanju aplikacije. Paketi, od katerih je aplikacija odvisna, morajo biti nameščeni na sistemu, če želimo, da aplikacija sploh deluje. Orodje `RubyGems` poskrbi, da se ti paketi namestijo (če je to potrebno) hkrati z našo aplikacijo.

Okolje Rails je primer sestavljenega paketa Ruby. Sestavljajo ga paketi, ki predstavljajo posamezne funkcionalnosti okolja. Podrobneje jih bomo spoznali v razdelku *Rails, Zagon aplikacije in spletnega strežnika*. Okolje Rails je mogoče tudi spremeniti ali razširiti z dodajanjem **paketov Rails**. V različici 3 okolja Rails ločimo dva tipa paketov Rails. Paketi, ki so popolnoma vključeni v okolje, so **standardni paketi okolja Rails** (`Railtie`). Standardni paketi okolja Rails imajo naslednje lastnosti:

- možnost ustvarjanja inicializatorjev, s katerimi so vključeni v zagon aplikacije
- možnost generiranja novih komponent okolja Rails z uporabo generatorjev
- možnost dodajanja dodatnih nastavitev v okolje Rails
- možnost uporabe pomožnih orodij standardnega paketa Rails `ActiveSupport`
- možnost dodajanja opravil orodju `Rake` (glej razdelek *Rails, Rake*)

Take pakete namestimo na sistem tako, da jih uporabljamo v vseh aplikacijah okolja Rails.
Preprosti paketi okolja Rails zgoraj naštetih lastnosti nimajo. Namestijo se znotraj določene aplikacije okolja v imenik `vendor/plugins/` in so uporabni le znotraj aplikacije. Lahko jih uporabimo za opravljanje določenih opravil, ki jih standardni paketi okolja Rails ne opravljajo. Ponavadi to pomeni, da za rešitev določene naloge v aplikaciji uporabimo že narejeno rešitev nekega drugega razvijalca.

V nadaljevanju bomo uporabljali termin **paket Ruby**, kadar se bo to nanašalo na knjižnico ali aplikacijo, nameščeno z orodjem `RubyGems`. **Standardni paket Ruby** ali **standardna knjižnica** Ruby je paket jezika Ruby s funkcijami, ki so del jezika Ruby, ali dodatne splošno uporabljane funkcije jezika Ruby. S terminom **paket Rails** bomo poimenovali vsak paket v okolju Rails. **Standardni paket Rails** je del samega okolja Rails. Vsi standardni paketi okolja Rails so tipa `Railtie`. Preproste dodatne pakete okolja Rails bomo ločili od paketov tipa `Railtie` z oznako **preprosti paket Rails**.

V nadaljevanju razdelka si bomo ogledali uporabo orodja `RubyGems`. Kot primer bomo namestili osnovni paket Rails. To storimo z ukazom:

```
gem install rails
```

S tem smo na sistem namestili osnovni paket okolja Rails. Na podoben način nameščamo tudi ostale standardne pakete jezika Ruby. Seznam vseh nameščenih paketov, skupaj s številko njihove različice, dobimo z ukazom:

```
gem list


*** LOCAL GEMS ***
```

```
abstract (1.0.0)
actionmailer (3.0.6)
actionpack (3.0.6)
activemodel (3.0.6)
...
rails (3.0.6)
railties (3.0.6)
rake (0.8.7)
...
RedCloth (4.2.7)
...
```

Razdelek bomo s tem zaključili. Predstavili smo orodje za nameščanje paketov Ruby in navedli primer nameščanja osnovnega paketa. Z izpisom seznama lahko preverimo različice standardnih paketov jezika Ruby, ki so nameščeni na sistemu.

2.4 Nizi in uporaba regularnih izrazov

Niz je objekt, ki vsebuje zaporedje znakov. Primer niza je denimo "`matematika`", pri čemer je narekovaj " rezerviran znak, ki določa začetek oziroma konec niza in ne spada v sam niz. V jeziku Ruby je niz implementiran v razredu `String`. V razdelku bomo obravnavali operacije, s katerimi ustvarjamo, sestavljamo, in spreminjamo nize. Pri razlagi metod razreda `String` bomo izkoristili znanje jezika Python tako, da bomo primerjali metode razreda `String` z metodami razreda `str`, ki je na voljo v Pythonu. Uporabo metod bomo prikazali na preprostih primerih. Razdelek je razdeljen na tri podrazdelke:

- *Ustvarjanje novega objekta razreda String*
  V razdelku bomo opisali, kako ustvarimo objekte razreda `String` v jeziku Ruby. Za vsak primer bomo dodali ustrezen primer z enakim učinkom, a napisanim v jeziku Python.
- *Metode razreda String*
  V razdelku bomo opisali metode razreda `String`, s katerimi sestavljamo in spreminjamo nize v jeziku Ruby. Tudi tu bomo pripravili še enakovredno različico v jeziku Python.
- *Uporaba regularnih izrazov v razredu String*
  V razdelku bomo opisali nekaj primerov metod razreda `String`, ki uporabljajo regularne izraze za iskanje delov nizov. Podali bomo tudi primer uporabe Ruby metode `sprintf` in jo primerjali z metodami za formatiranje nizov, ki so na voljo v jeziku Python.

2.4.1 Ustvarjanje novega objekta razreda String

Obstaja več načinov s katerimi lahko ustvarimo objekt razreda `String`. Prvi način sledi iz dejstva, da je Ruby objektno orientiran jezik. Če hočemo ustvariti nov objekt nekega razreda, to storimo z uporabo metode razreda `new`:

```
ruby>s = String.new("abrakadabra")
=> "abrakadabra"
```

Poklicali smo konstruktor razreda `String`. Kot parameter smo mu podali niz "`abrakadabra`" ter objekt poimenovali in shranili. V jeziku Ruby vsi razredi dedujejo iz osnovnega razreda `Object` ali razreda, ki deduje iz tega razreda. V jeziku Python pa obstajajo vgrajeni standardni razredi, ki dedujejo iz razreda `type`. Razred `str` je eden od njih. Če želimo narediti enako kot zgoraj v jeziku Python, moramo izvesti:

```
python> s = str.new(str,"abrakadabra")
>>> s
'abrakadabra'
```

S konstruktorjem razreda `str` smo poklicali konstruktor razreda `type` in mu kot prvi parameter podali `str` – tip standardnega razreda, ki ga ustvarjamo, kot drugi parameter pa niz znakov. Seveda obstajajo tako v jeziku Ruby kot v jeziku Python precej lažji načini za ustvarjanje nizov. Denimo:

```
ruby>s = "abrakadabra"
=> "abrakadabra"
```

Oba jezika poznata dinamično določanje tipov objektov, tako da je ukaz v jeziku Python enak:

```
 python> s = "abrakadabra"
>>> s
'abrakadabra'
```

V Rubyju lahko tako kot pri Pythonu namesto dvojnih narekovajev pri določanju začetka in konca niza uporabimo enojne narekovaje:

```
ruby>s = 'abra\nkadabra'
ruby> puts s
abra\nkadabra
ruby>s = “abra\nkadabra”
ruby>puts s
abra
kadabra
```

V niz smo vrinili kombinacijo znakov `\n`, ki jo, če uporabimo enojne narekovaje, ohranimo tako kot je. Če pa uporabimo dvojne narekovaje, se ta kombinacija tolmači kot prehod v novo vrstico. Za izpis smo uporabili metodo `puts`, ki je ena od metod modula `Kernel`, ki jih Ruby uporablja za izpis niza na standardni izhod. V jeziku Python take razlike med dvojnimi in enojnimi narekovaji ni. Enak učinek kot v Rubyju bi dosegli z uporabo določila `r (raw)`, s čimer dosežemo, da se znaki v narekovajih ne interpretirajo, ali z uporabo leve poševnice pred posebnim znakom:

```
python> s = r"abra\nkadabra"
>>> print(s)
abra\nkadabra
>>> s = "abra\\nkadabra"
>>> print(s)
abra\nkadabra
```

V jeziku Ruby poznamo še več pripravnih načinov za kreiranje nizov. Uporabimo lahko na primer operatorja `%` in <<:

```
ruby>s = %!abrakadabra!
```

Tudi zgornji ukaz naredi objekt razreda `String`. Za označitev začetka in konca niza uporabimo poljuben znak, ki ni črka ali številka. V našem primeru smo uporabili `!`. Operator `%` lahko uporabimo tudi v kombinaciji z znakom, ki označuje vrsto niza. Z uporabo `%Q` naredimo niz z dvojnimi narekovaji. Z uporabo `%q` naredimo niz z enojnimi narekovaji. Z uporabo `%x` naredimo niz z narekovaji nazaj, ki je primeren za shranjevanje izpisa nekega ukaza.
Operator << je zelo primeren za vpisovanje celih stavčnih blokov v nize:

```
ruby>s = <<stavek
```

```
"Prva vrstica
"Druga vrstica
"Zadnja vrstica
"stavek
=> "Prva vrstica\nDruga vrstica\nZadnja vrstica\n"
```

S tem smo v niz `s` vstavili cel odstavek. Z besedo stavek (seveda jo lahko nadomesti katerakoli druga) smo označili začetek in konec bloka.V Pythonu podobno dosežemo z uporabo trojnih narekovajev:

```
python> s = """Prva vrstica
... Druga vrstica
... Tretja vrstica"""
>>> s
'Prva vrstica\nDruga vrstica\nTretja vrstica'
```

V Rubyju operator << lahko uporabimo tudi tako, da določimo, s kakšnimi narekovaji se bo kreiral niz. To naredimo da znački začetka `stavek` dodamo narekovaje. Uporabimo << `"stavek"`, če želimo dvojne narekovaje, `<<'stavek'` za enojne, `<<`stavek`` za narekovaje nazaj. Tako z

```
ruby>s = <<`stavek`
“Prva vrstica
“Druga vrstica
" Zadnja vrstica
"`stavek`
```

v `s` dobimo:

```
`Prva vrstica\nDruga vrstica\nZadnja vrstica\n`
```

Če želimo v stavku presledek na začetku, uporabimo `<<-stavek`.

S tem smo končali razdelek o kreaciji nizov v Rubyju. V naslednjem razdelku se bomo posvetili nekaterim metodam razreda `String`. Opisali bomo metode, s katerimi nize združujemo in spreminjamo, ter metodo `split`.

2.4.2 Metode razreda String

V tem razdelku si bomo ogledali nekaj metod, s katerimi nize združujemo, spreminjamo, in vračamo posamezne dele vsebine nizov. Na koncu razdelka bomo predstavili še metodo `split`. Ta služi za pretvarjanje vsebine niza v tabelo nizov. Metoda `split` obstaja v obeh jezikih, jeziku Ruby in jeziku Python. Metodi se v delovanju nekoliko razlikujeta in ogledali si bomo razlike med njima.
Najprej si oglejmo združevanje dveh nizov. To se v Rubyju da doseči na več načinov. V zgledu spodaj smo predstavili tri različne primere združevanja. V prvem primeru smo uporabili operator `+` razreda `String`, v drugem primeru operator `<<`, v tretjem primeru pa poklicali metodo `str.concat`:

```
ruby> s = "abra" + "kadabra"
=> "abrakadabra"
ruby> s = "abra" << "kadabra"
=> "abrakadabra"
```

```
ruby> s = "abra".concat("kadabra")

=> "abrakadabra"
```

Tudi v Pythonu obstaja več načinov združevanja nizov. Prvi primer je enak kot v Rubyju. V drugem primeru smo niza združili z uporabo formatov. Tretji zgled pa nima ekvivalenta, saj so v Pythonu nizi nespremenljivi, torej obstoječih nizov ne moremo spreminjati:

```
python> s = "abra" + "kadabra"

>>> s

'abrakadabra'


python> s = "%s%s" %("abra", "kadabra")

>>> s

'abrakadabra'
```

Za vračanje posameznih delov nizov v Rubyju uporabljamo več metod.
Najprej si pripravimo testni objekt. To naredimo v obeh jezikih enako:

```
ruby & python>s = "abrakadabra"
```

Če želimo le del "`bra`", ga lahko dobimo takole:

```
ruby> s['bra']

=> "bra"
```

Če želimo le prvi znak, pa izvedemo:

```
ruby> s[0].chr

=> "a"
```

Če želimo znake od 1 do 3, izvedemo:

```
ruby> s[1..3]

=> "bra"
```

V Pythonu bi za enake rezultate uporabili naslednje metode:

```
python> s[1:4]

'bra'

python> s[0]

'a'
```

V Rubyju lahko niz tudi spremenimo z ukazom:

```
ruby>s["bra"] = "arb"

=> "arb"

ruby> s

=> "aarbkadabra"
```

V Pythonu tega ne moremo, ker je niz nespremenljiv objekt. Lahko pa v Rubyju niz zamrznemo in s tem dosežemo, da se ga ne da spremeniti:

```
ruby> s.freeze

=> "abrakadabra"

> s["bra"] = "arb"

TypeError: can't modify frozen string
```

Dejstvo, da so nizi nespremenljivi, je morda glavna razlika med Pythonovim razredom `str` in Rubyjevim razredom `String`.
Oglejmo si še metodo `split`, ki jo poznata oba jezika. Metoda služi za pretvarjanje niza v tabelo znakov ali skupin znakov, vsebovanih v podanem nizu. V Rubyju izvedemo:

```
ruby> "abrakadabra".split(//)
=> ["a", "b", "r", "a", "k", "a", "d", "a", "b", "r", "a"]

ruby> "abrakadabra".split()
=> ["abrakadabra"]

ruby> "abrakadabra".split(/a/)
=> ["", "br", "k", "d", "br"]
```

Prva stvar, ki jo opazimo, je, da se da metodo klicati na nepoimenovanem objektu, kar je možno tudi v Pythonu. V Pythonu pa se ne da izvesti prvega primera, torej uvrstiti vse znake v tabelo z uporabo praznega znaka kot separatorja.

```
python> "abrakadabra".split()
['abrakadabra']

python> "abrakadabra".split('a')
['', 'br', 'k', 'd', 'br', '']
```

Seveda obstaja v obeh razredih, ki predstavljata nize, še veliko metod, ki jih tukaj nismo opisali. Naš namen je bil le predstaviti nekatere razlike med implementacijami niza v obeh jezikih. V naslednjem razdelku bomo spoznali uporabo regularnih izrazov v jeziku Ruby, ki se uporabljajo za iskanje delov niza. Na koncu razdelka si bomo ogledali še metodo `sprintf` in jo primerjali z metodo `str.format` jezika Python.

2.4.3 Uporaba regularnih izrazov v razredu String

Uporabo regularnih izrazov si bomo ogledali v nekaj primerih. V teh primerih bomo iskali dele daljših nizov.
Najprej naredimo testni niz, pri čemer bomo uporabili znanje, ki smo ga osvojili v prejšnjem razdelku:

```
ruby> s = <<"imenik"
" Peter (041)571-678
" Mojca (01)536789
" Janez (05)1645389
" Lojze ne vem
" imenik
=> "Peter (041)571-678\nMojca (01)536789\nJanez (05)1645389\nLojze ne
vem\n"
```

V Pythonu naredimo enak niz z ukazom:

```
python> s = """ Peter (041)571-678
... Mojca (01)536789
... Janez (05)1645389
```

```
... Lojze ne vem """
>>> s

' Peter (041)571-678\nMojca (01)536789\nJanez (05)1645389\nLojze ne
vem\n'
```

Oglejmo si sedaj v Rubyju iskanje vseh vnosov, ki vsebujejo telefonske številke. Izvedemo ukaz:

```
ruby> s.grep(/\(\d{2,3}\)\d{3,4}-?\d{3}$/)

=> ["Peter (041)571-678\n", "Mojca (01)536789\n", "Janez (05)1645389\n"]
```

Ta ukaz nam vrne vse vrstice, ki ustrezajo regularnemu izrazu `\(\d{2,3}\)\d{3,4}-?\d{3}$`. Če želimo podobne podatke dobiti v Pythonu, moramo najprej naložiti Pythonov modul za regularne izraze z imenom `re`. Nato izvedemo:

```
python> import re

 >>> m = re.findall('\w* \(\d{2,3}\)\d{3}-?\d{3}', s)

>>> m

['Peter (041)571-678', 'Mojca (01)536789', 'Janez (05)164538']
```

Če bi uporabili enak regularni izraz kot v Rubyju, bi dobili le telefonske številke. Ukaz `findall` namreč vrača le tiste dele niza, ki ustrezajo vzorcu. Ukaz `grep` v Rubyu pa poišče in vrne vrstice, ki vsebujejo vzorec. V našem primeru ko želimo v Pythonu dobiti poleg številke še ime, smo v regularni izraz morali dodati še `\w*`, ki "pobere" del pred telefonsko številko. Dejstvo, da Python pri uporabi `findall` vrača le tiste dele niza, ki ustrezajo vzorcu, je seveda pogosto točno to, kar želimo. Vendar tudi Ruby pozna metode, s katerimi vračamo le tisti del niza, ki ustreza regularnemu izrazu:

```
ruby>s.scan(/\w* \(\d{2,3}\)\d{3,4}-?\d{3}$/)[0..2]

=> ["Peter (041)571-678", "Mojca (01)536789", "Janez (05)1645389"]
```

Uporabili smo isti regularni izraz kot v Pythonu in dobili isti rezultat. Razlika je le v tem, da nam v Rubyju ni bilo treba vključevati nobenega posebnega modula, da bi lahko uporabljali regularne izraze. Ta je namreč naložen avtomatsko, ko ustvarimo objekt razreda `String`.
Poleg zgoraj uporabljenih metod omenimo še dva operatorja. Z njima preverjamo, če niz vsebuje regularni izraz. To sta =~ in `!~` . Operator =~ vrača indeks znaka, ki začenja regularni izraz v nizu. Če v nizu ni podniza, ki bi ustrezal regularnemu izrazu, vrne vrednost `nil`. Operator `!~` pa vrača vrednost `true`, če regularnega izraza ni v nizu in `false`, če le-ta v nizu obstaja.
S temi kratkimi primeri smo zaključili obravnavo uporabe regularnih izrazov. Sedaj si bomo ogledali še metodo jezika Ruby za izpis niza `sprintf` in jo primerjali s Pythonovimi možnostmi formatiranja nizov.
Defiirajmo naslednje podatke v Rubyju in Pythonu:

```
ruby & python>ime = "Andrej"

      priimek = "Kovic"

visina = 181

teza = 89
```

V niz `podatki` bomo spravili zgornje podatke in izračunani masni indeks.
V Pythonu poženemo ukaz:

```
python> podatki = "ime: {0}, priimek: {1}, visina: {2}, teza: {3}, masni
indeks: {4}".format(ime, priimek, visina, teza, teza/((visina/100)**2))

>>> podatki

'ime: Andrej, priimek: Kovic, visina: 181, teza: 89, masni indeks:
27.1664479106'
```

V Rubyju pa:

```
ruby> podatki = sprintf("ime: %s, priimek: %s, visina: %d, teza: %d
```

```
masni indeks: %f", ime, priimek, visina, teza, teza/((visina/100.0)**2))
=> "ime: Andrej, priimek: Kovic, visina: 181, teza: 89 masni indeks:
27.166448"
```

Pri Rubyju smo morali zaradi zaokroževanja uporabiti `100.0`, v Pythonu pa to ni bilo potrebno. Seveda bi v Pythonu lahko uporabili tudi drugačen ukaz:

```
podatki = "ime: %s, priimek: %s, visina: %d, teza: %d, masni indeks: %f"
%(ime,priimek, visina, teza, teza/((visina/100)**2))

>>> podatki

'ime: Andrej, priimek: Kovic, visina: 181, teza: 89, masni indeks:
27.166448'
```

Ta ukaz je bolj podoben zgornjemu primeru uporabe ukaza `sprintf` v Rubyju. Ruby razred `String` nima metode, ki bi bila podobna Pythonovi metodi `str.format`.

S tem primerom smo zaključili poglavje Nizi in regularni izrazi. Spoznali smo metode jezika Ruby s katerimi ustvarimo nize, jih združujemo in spreminjamo. Ogledali smo si metodo `split`, s katero niz spremenimo v tabelo znakov. Pri iskanju vzorcev v daljših nizih smo uporabili regularne izraze. No koncu poglavja smo spoznali še metodo `sprintf`, ki jo v Rubyju uporabljamo za sestavljanje nizov po določenem formatu. V naslednjem poglavju bomo spoznali uporabo pogojnih ukazov v jeziku Ruby.

2.5 Pogojni stavki in zanke

V razdelku bomo opisali pogojne stavke in zanke. Začeli bomo s primeri uporabe pogojnih stavkov `if` in `unless`. Nato bomo opisali nekaj primerov uporabe zank `for`, `while` in `until`. Ogledali si bomo primer uporabe operatorja `?`. Na koncu razdelka bomo predstavili poseben primer uporabe pogojnega stavka `unless`. Uporabimo ga namreč lahko kot spreminjevalca enovrstičnega ukaza (`statement modifier`). V tem primeru se ukazi na začetku vrstice pred ključno besedo `unless` obravnavajo kot blok ukazov, ki se izvede ob izpolnjenem (ali v primeru `unless` neizpolnjenem) pogoju v pogojnem stavku ali zanki. Pogojni stavek `if` in zanko `while` namreč lahko prav tako kot pogojni stavek `unless` uporabimo kot spreminjevalca enovrstičnega ukaza.
Večini primerov bomo poiskali ustrezne primere z enakim učinkom v jeziku Python.
Začeli bomo z naslednjim primerom uporabe pogojnega stavka `if`:

```
ruby>if a > 0
       puts "Pogoj izpolnjen: a > 0"
     elsif a == 0
       puts "Pogoj izpolnjen: a == 0"
     else
       puts "Pogoj ni izpolnjen: a < 0"
     end
```

V preprostem sestavljenem primeru preverjamo število `a`. V primeru, da je število večje od 0, je izpolnjen prvi pogoj `if a > 0`, v primeru, ko je enako 0, je izpolnjen drugi pogoj `elsif a == 0` . Če pa ni izpolnjen noben od naštetih pogojev, se izvede koda v bloku ukazov pogojnega stavka `else`. V Pythonu enak učinek dosežemo s praktično skoraj enako kodo:

```
python> if a > 0:
...   print ("Pogoj je izpolnjen: a > 0")
... elif a ==0:
```

```
... print ("Pogoj je izpolnjen: a == 0")
... else:
...    print("Pogoj ni izpolnjen: a < 0")
```

Ukaz `unless` je nasproten ukazu `if`. Koda v bloku ukazov pogojnega stavka `unless` se izvede, kadar pogoj v pogojnem stavku ni izpolnjen:

```
ruby> a = 5
=> 5
 unless a > 6
    puts "Pogoj ni izponjen: a = 5"
 end
Pogoj ni izponjen a = 5
```

V Pythonu enak učinek dosežemo z naslednjo kodo:

```
python> a = 5
>>> if not (a > 6):
...   print ("Pogoj ni izpolnjen: a = 5")
...
Pogoj ni izpolnjen: a = 5
```

Ogledali smo si primere uporabe pogojnih stavkov `if` in `unless`. Sedaj si bomo ogledali še primere uporabe zank `for`, `while` in `until`. Kot vemo, zanke uporabljamo za izvajanje bloka ukazov, dokler je pogoj v zanki izpolnjen.
Oglejmo si preprost primer uporabe zanke `while`:

```
ruby> while (i < 3)
              puts i
              i += 1
             end
0
1
2
```

Enak učinek dosežemo v Pythonu z naslednjo kodo:

```
python> a = 0
>>> while (a < 3):
...    print("%d" %a)
...    a += 1
...
0
1
2
```

Enak učinek bi v Rubyju dosegli tudi z uporabo zanke `for`

```
ruby> for a in 0..2
              puts a
          end
```

```
0

1

2
```

ki obstaja tudi v Pythonu:

```
python> for i in range(3):
...             print(i)
...
0
1
2
```

Pri Rubyju pa poznamo še zanko `until`:

```
ruby> a = 0

 until a > 2

    puts a

    a += 1

 end

0

1

2
```

Z zanko `until` torej izvajamo blok ukazov, dokler pogoj v zanki `until` ni izpolnjen.
Sedaj si bomo ogledali še primer uporabe operatorja `?`.
Uporabljamo ga za določanje vrednosti spremenljivk. Če je pogoj izpolnjen, spremenljivka dobi vrednost izraza za operatorjem `?`. Če pa pogoj ni izpolnjen, vanjo shranimo vrednost izraza za znakom `:`.
Za boljšo predstavo si oglejmo naslednji primer :

```
ruby> a = 0

=> 0

ruby> b = (a > 0) ? 1 : 0

=> 0

ruby> b

=> 0

ruby> a = 1

=> 1

ruby> b = (a > 0) ? 1 : 0

=> 1

ruby> b

=> 1
```

Vidimo, da se vrednost spremenljivke `b` določa v odvisnosti od spremenljivke `a`. Kadar je vrednost v `a` enaka 0, je tudi vrednost v `b` enaka 0. V nasprotnem primeru pa spremenljivka `b` dobi vrednost 1. Seveda bi isto lahko dosegli tudi s pogojnim stavkom, a večkrat je uporaba operatorja `?:` bolj pregledna. No, tudi Python pozna podoben operator, le da se zapiše nekoliko drugače:

```
python> a = 0

>>> b = 1 if (a > 0) else 0

>>> b
```

```
>>> a = 1

>>> b = 1 if (a > 0) else 0

>>> b

1
```

Na primeru bomo spoznali še zanimivo kompleksno uporabo pogojnega stavka `unless`.
Kot smo omenili že v uvodu, obstaja (tako kot pri pogojnem stavku `if` in zanki `while`) še en način uporabe tega pogojnega stavka. Lahko ga uporabimo kot spreminjevalca enovrstičnega ukaza (`statement modifier`):

```
> puts "Pogoj ni izpolnjen" unless (2 == 2)

=> nil

> puts "Pogoj ni izpolnjen" unless  (2 == 3)

Pogoj ni izpolnjen

=> nil
```

Ukaz na začetku stavka `puts "Pogoj ni izpolnjen"` se torej izvede le v primeru, ko pogoj ni izpolnjen. To, da se ukaz ali blok ukazov izvede, kadar pogoj ni izpolnjen, je, kot vemo, značilnost pogojnega stavka `unless`. Bistvena razlika med tem in osnovnim primerom uporabe je le v tem, da se v običajnem primeru blok ukazov nahaja za pogojem, v posebnem primeru pa je vrinjen med pogoj in začetek vrstice.
Za konec si oglejmo še umeten, na splošno neuporaben zgled. Ogledali si ga bomo zato, ker je logika zaporedja izvrševanja ukazov zelo podobna primeru, ki ga bomo srečali v razdelku *Rails, Inicializacija aplikacije in zagon spletnega strežnika*:
V datoteki `primer_unless.rb` imamo sledečo kodo:

```
#!/usr/bin/env ruby

module PrimerUnless

  def self.test_unless!

     puts "Ali naj bo pogoj izpolnjen?[D/N]"

     vrni = gets # preberemo znak

     pogoj = false

     if vrni.chomp.eql?("D") # je znak enak znaku "D"

        pogoj = true

     end

     return unless pogoj==false

     puts "PrimerUnless: Pogoj ni bil izpolnjen in return se ni izvedel"

     exec "echo", "In izvede se novi ukaz..."

  end

end
```

V datoteki je definiran modul `PrimerUnless` in metoda modula `test_unless!` (glej razdelek *Ruby, Razredi, metode in moduli*). Omenimo le, da `!` na koncu metode pomeni, da je metoda potencialno destruktivna. To pomeni, da se ob klicu metode dejansko lahko spremenijo vrednosti spremenljivk objekta razreda, na katerem je klicana.
Zanima nas, ali se `return` izvede v vsakem primeru, ali pa le takrat, ko pogoj ni izpolnjen. V ta namen si pripravimo še skripto `preveri_return_unless.rb`, ki bo naložila modul in izvedla `test_unless!`:

```
#!/usr/bin/env ruby
class Test
   def test_return
       require "primer_unless.rb"
      PrimerUnless.test_unless!
      puts "Return se je izvedel. Nadaljujemo z
           izvajanjem skripte Test..."
   end
end
Test.new.test_return
```

V prvem delu skripte definiramo razred `Test`. Skripta ob izvajanju v zadnji vrstici ustvari objekt razreda `Test` in na njem izvede metodo `test_return`. Ta z ukazom `require "primer_unless.rb"` naloži modul `PrimerUnless`. To nam omogoči, da lahko v metodi `test_return` izvajamo metode, ki so v modulu `PrimerUnless` definirane. V naslednji vrstici `PrimerUnless.test_unless!` metode `test_return` torej pokličemo metodo `test_unless!`, s katero bomo preverili obnašanje našega programa ob uporabi pogojnega stavka `unless` v obliki spreminjevalca enovrstičnega ukaza.
Poglejmo rezultate:

```
root@andrej-desktop:~/ruby/rubytest# ./preveri_return_unless.rb
Ali naj bo pogoj izpolnjen?[D/N]
N
PrimerUnless: Pogoj ni bil izpolnjen in return se ni izvedel
In izvede se novi ukaz...
```

Vidimo, da se je izvedla metoda `PrimerUnless.test_unless!`, `return` pa očitno ne. Če bi se izvedel, bi dobili izpis `"Return se je izvedel. Nadaljujemo z izvajanjem skripte Test..."` Ob izvedbi ukaza return bi se namreč metoda `test_unless!` zaključila in izvajanje programa bi se nadaljevalo v vrstici za klicem metode, torej v naslednji vrstici metode `test_return`. To bi se sicer zgodilo tudi drugače ob izteku metode `test_unless!`, vendar smo v zadnji vrstici kode metode `test_unless!` uporabili majhen trik. Napisali smo namreč:

```
exec "echo", "In izvede se novi ukaz..."
```

Z ukazom `exec` modula `Kernel` ustavimo delovanje procesa naše skripte, v katerem se trenutno izvajata metodi `test_return` in iz nje klicana metoda `PrimerUnless.test_unless!`. Na mestu našega procesa bo metoda `exec` izvedla sistemski ukaz `echo "In izvede se novi ukaz"`, ki povzroči izpis teksta na zaslon. S tem dodatkom smo torej preprečili nadaljnje izvajanje skripte `test_return`.
Po logiki povsem enak primer uporabe `unless` bomo videli v razdelku *Rails, Inicializacija aplikacije in zagon spletnega strežnika*.
V našem primeru nam preostane le še, da preverimo, kaj se zgodi, ko je pogoj izpolnjen:

```
root@andrej-desktop:~/ruby/rubytest# ./preveri_return_unless.rb
Ali naj bo pogoj izpolnjen?[D/N]
D
Return se je izvedel. Nadaljujemo z izvajanjem skripte Test...
```

Return se je očitno izvedel in izvajanje metode `test_unless` se je zaključilo. Izpis potrjuje nadaljevanje izvajanja kode v metodi `test_return`.

2.6 Tabele in slovarji

Tabela je v jeziku Ruby objekt razreda `Array` in predstavlja zbirko objektov. Vsak objekt v tabeli je element tabele. Vsakemu elementu pripada indeks, ki označuje njegov položaj v tabeli. Prvi element v tabeli je označen z indeksom 0.
Tudi slovar, ki je objekt razreda `Hash`, je zbirka objektov. Vendar so le-ti shranjeni v drugačnem formatu kot pri tabeli. Vsak element slovarja je sestavljen iz ključa in njegove vrednosti. V jeziku Ruby lahko tako za ključe kot za njihove vrednosti uporabimo objekt katerega koli razreda, definiranega v jeziku Ruby.
Razdelek je razdeljen na dva logična sklopa:

- *Tabele*
  V razdelku si bomo ogledali nekaj metod razreda `Array`, s katerimi tabele ustvarjamo ter elemente tabele dodajamo in brišemo. Na koncu si bomo ogledali še metode, s katerimi se sprehajamo po vseh elementih tabele in na njih izvajamo določene operacije.Vsak primer bomo podkrepili s primerom z enakim pomenom v jeziku Python.
- *Slovarji*
  V razdelku bomo spoznali nekaj metod razreda `Hash`, ki jih uporabljamo za ustvarjanje slovarjev, ter za dostopdo elementov slovarjev in njihovo dodajanje in brisanje. Tudi tu bomo za vsak primer navedli še takega z enakim pomenom v jeziku Python.

2.6.1 Tabele

V uvodu smo izvedeli, da je tabela objekt razreda `Array`. Novo tabelo najlažje ustvarimo tako, da med oglatimi oklepaji navedemo elemente te tabele:

```
ruby> tabela = [1,2,"tri",4]

=> [1, 2, "tri", 4]
```

Kot vidimo, lahko v tabeli uporabimo različne objekte razreda jezika Ruby. V našem primeru smo uporabili le števila in nize.
Tabelo bi lahko ustvarili tudi z direktno uporabo konstruktorja razreda `Array`:

```
ruby> tabela2 = Array.new(4)

=> [nil, nil, nil, nil]
```

S tem smo ustvarili le tabelo, katere elementi pa so ničelni objekti, oz. `nil`. Te elemente lahko kasneje zamenjamo s pravimi.
V konstruktorju pa poleg velikosti tabele kot drugi parameter lahko podamo blok ukazov, s katerim generiramo elemente:

```
ruby> tabela3 = Array.new(4){|i| 2*i+1}

=> [1, 3, 5, 7]
```

Sestavili smo sestavili tabelo lihih števil velikosti 4.
Tudi v Pythonu novo tabelo definiramo na enak način z uporabo oglatih oklepajev:

```
python> tabela = [1,2,"tri",4]

>>> tabela

[1, 2, 'tri', 4]
```

Tabela je v Pythonu objekt razreda `list`. Konstruktor tabele pa je drugačen

```
python> tabela = list([1,2,"tri",4])

>>> tabela

[1, 2, 'tri', 4]
```


saj mu lahko podamo tudi elemente tabele, kar pri Rubyju ni bilo mogoče.

Omenimo še, da v Pythonu poleg tabel poznamo še nabore. Nabori elementov so označeni z okroglimi oklepaji `()`, njihova posebnost pa je, da so nespremenljivi. Tudi pri Rubyju lahko dosežemo, da se tabela "vede" enako kot nabor v jeziku Python. Z uporabo metode `freeze` postane tabela nespremenljiva:

```
ruby> tabela = [1,2,"tri",4]
=> [1, 2, "tri", 4]
ruby> tabela.freeze
=> [1, 2, "tri", 4]
ruby> tabela << 5
TypeError: can't modify frozen array
        from (irb):4:in `<<'
        from (irb):4
        from :0
```

Sedaj se bomo posvetili dodajanju in odvzemanju elementov iz tabele. Tabelo bomo v obeh jezikih uporabili kot sklad. Najprej si pripravimo prazno tabelo:

```
python & ruby> tabela = []
```

Nato izvedemo v Rubyju naslednje ukaze:

```
ruby> tabela << "prvi"
=> ["prvi"]
> tabela << "drugi"
=> ["prvi", "drugi"]
> tabela.pop
=> "drugi"
> tabela
=> ["prvi"]
```

Tabelo smo z operatorjem << na konec najprej dodali element "`prvi`", nato še element "`drugi`". Z metodo `pop` smo iz tabele nato odstranili element "`drugi`". Pri Pythonu za povsem enak rezultat izvedemo ukaze:

```
python> tabela.append("prvi")
>>> tabela.append("drugi")
>>> tabela
['prvi', 'drugi']
>>> tabela.pop()
'drugi'
>>> tabela
['prvi']
```

Za dodajanje elementa na konec tabele smo uporabili `append`, za odstranjevanje pa prav tako kot pri Rubyju metodo `pop`. V obeh primerih vidimo, da smo tabelo uporabili kot sklad. Zadnji element vedno dodamo na vrh sklada. Z metodo `pop` pa odstranimo element z vrha sklada.

V obeh jezikih obstaja še vrsta metod, s katerimi lahko elemente vstavljamo na poljubna mesta v tabeli, ali po potrebi tudi spreminjamo. Teh metod ne bomo posebej opisovali. Opisali pa bomo nekaj metod, s katerimi se sprehajamo po seznamu elementov tabele.

V obeh jezikih lahko za izpis posameznih elementov tabele uporabimo izpis v zanki (glej razdelek *Ruby, Pogojni stavki in zanke*). V Rubyju to storimo na primer takole:

```
ruby> for  i in 0...tabela.size do
        puts tabela[i]
        i = i+1
      end
prvi
drugi
tretji
=> nil
```

V Pythonu pa izvedemo ukaze:

```
python> for i in range(len(tabela)):
...       print(tabela[i]);
...
prvi
drugi
tretji
```

V obeh primerih smo elemente izpisali z uporabo indeksov. Kot smo omenili, ima prvi element indeks 0, zadnji pa indeks dolžina tabele - 1.
V Rubyju pa poznamo še nekaj metod, ki nam precej olajšajo delo z vsemi elementi v tabeli. Oglejmo si funkcijo `each`:

```
ruby> tabela.each { |elt| puts elt}
prvi
drugi
tretji
=> ["prvi", "drugi", "tretji"]
```

Vidimo, da smo dosegli enak izpis kot v zgornjih dveh primerih. Metodo `each` pokličemo z blokom ukazov (glej razdelek *Ruby, Bloki in procedure*). Ta blok ukazov se nato izvede za vsak element v tabeli. Ime tega elementa določimo v prvem delu bloka, med znakoma | |. V našem primeru smo posamezen element preprosto izpisali. Podoben učinek bi dosegli v Pythonu z:

```
python> for elt in tabela :
          print(elt)
```

Razlika med jezikoma je v tem, da nam v Rubyju metoda poleg tega, da opravi predpisano akcijo, še vrne tabelo, na kateri je bila metoda `each` izvedena. Metoda `each` nam vedno vrne nespremenjeno tabelo.
Če želimo tabelo z blokom ukazov spremeniti, moramo uporabiti metodo `map`. Oglejmo si preprost primer, v katerem bomo vsak element v tabeli podvojili:

```
ruby> tabela.map { |elt| elt = elt*2}
=> ["prviprvi", "drugidrugi", "tretjitretji"]
```

V Rubyju obstaja še cela vrsta metod za delo z objekti razreda `Array`. V tem razdelku smo si ogledali le tiste, ki jih bomo uporabili v naslednjih razdelkih. Te metode so pogosto uporabljane, ker predstavljajo zelo enostaven način za izvajanje operacij, ki jih ponavadi izvajamo v zankah.

2.6.2 Slovarji

Kot smo povedali v uvodu razdelka 2.6, je tudi slovar zbirka objektov jezika Ruby. Vsak objekt slovarja je sestavljen iz objekta, ki predstavlja ključ in objekta, ki predstavlja vrednost. Pri Rubyju lahko tako za ključ, kot za vrednost, uporabimo katerikoli objekt. Ključi slovarja morajo seveda biti enolično določeni. Z uporabo posameznega ključa dostopamo do vrednosti, ki je pod tem ključem shranjena v slovarju. Slovar je v jeziku Ruby objekt razreda `Hash`.
Na primeru si oglejmo, kako v Rubyju definiramo nov slovar:

```
ruby> slovar = { "prvi" => "prva vrednost", 2 => "druga vrednost", 3 =>
Array.new(3,"A") }

=> {"prvi"=>"prva vrednost", 2=>"druga vrednost", 3=>["A", "A", "A"]}

> puts slovar[3]

A

A

A

=> nil

> puts slovar["prvi"]

prva vrednost

=> nil

> puts slovar[2]

druga vrednost
```

Vidimo, da smo za prvi ključ uporabili niz "`prvi`", za drugega in tretjega pa števili 2 in 3. Vrednosti prvih dveh ključev sta niza "`prva vrednost`" in "`druga vrednost`", za tretjo vrednost pa smo uporabili kar objekt nove tabele. S tem smo prikazali, da lahko tako za ključ kot vrednost izberemo objekt kateregakoli razreda jezika Ruby. Tudi v Pythonu lahko tako za ključe slovarjev kot tudi vrednosti ključev izbiramo poljubne objekte jezika Python:

```
python> slovar = { "prvi":"prva vrednost", 2:"druga vrednost",
3:list(['A','B','C']) }

>>> slovar

{2: 'druga vrednost', 3: ['A', 'B', 'C'], 'prvi': 'prva vrednost'}

>>> slovar[3]

['A', 'B', 'C']

>>> slovar[2]

'druga vrednost'

>>> slovar["prvi"]

'prva vrednost'
```

Oglejmo si še, kako v slovar dodamo nov element. V obeh jezikih to storimo tako, da enostavno navedemo nov par ključa in vrednosti:

```
ruby & python> slovar["cetrti"] = "cetrta vrednost"

>>> slovar

{'cetrti': 'cetrta vrednost', 2: 'druga vrednost', 3: ['A', 'B', 'C'],
'prvi': 'prva vrednost'}
```

Posamezen element slovarja lahko tudi pobrišemo. V Rubyju to storimo z ukazom `delete`. Poglejmo zgled:


```
ruby> slovar
```

```
=> {"cetrti"=>"cetrta vrednost", "prvi"=>"prva vrednost", 2=>"druga
vrednost", 3=>["A", "A", "A"]}

> slovar.delete("cetrti")

=> "cetrta vrednost"

> slovar

=> {"prvi"=>"prva vrednost", 2=>"druga vrednost", 3=>["A", "A", "A"]}
```

V Pythonu pa izvedemo ukaz:

```
>>> del slovar["cetrti"]

>>> slovar

{2: 'druga vrednost', 3: ['A', 'B', 'C'], 'prvi': 'prva vrednost'}
```

Poglejmo še uporabo metode `each`, ki jo v Rubyju, podobno kot v tabeli, lahko uporabimo tudi v slovarju. Oglejmo si primer:

```
ruby> slovar.each { |key, value| puts "Ključ: " + key.to_s + " Vrednost:
" + value.to_s }

Ključ: prvi Vrednost: prva vrednost

Ključ: 2 Vrednost: druga vrednost

Ključ: 3 Vrednost: AAA

=> {"prvi"=>"prva vrednost", 2=>"druga vrednost", 3=>["A", "A", "A"]}
```

Metodo `each` smo poklicali z blokom ukazov. Metoda `each` posreduje bloku ukazov parametra `key` in `value`. Prvi predstavlja ključ posameznega elementa slovarja, drugi pa njegovo vrednost. Ukazi v bloku ukazov se izvedejo na vseh elementih slovarja.
Podobno dosežemo v Pythonu z:

```
python> for key, value in slovar.items():

...    print("Ključ: %s Vrednost: %s" %(key,value))

...

Ključ: 2 Vrednost: druga vrednost

Ključ: 3 Vrednost: ['A', 'B', 'C']

Ključ: prvi Vrednost: prva vrednost
```

S tem smo zaključili kratek pregled tabel in slovarjev. Ogledali smo si metode, s katerimi tabele in slovarje ustvarjamo, jim dodajamo in odstranjujemo elemente. Za vsak tak primer smo dodali ustrezen primer v jeziku Python. Na koncu vsakega razdelka smo si ogledali še uporabo metode `each`, s katero v Rubyju izvajamo blok ukazov na vseh elementih tabele ali slovarja.

2.7 Razredi, metode in moduli

V razdelku *Ruby, Lastnosti* smo izvedeli, da vse, kar definiramo v Rubyju, predstavlja objekt. V tem razdelku bomo predstavili, kako v Rubyju definiramo razrede, metode razredov in metode njihovih objektov. Predstavili bomo posebne metode razredov, ki se imenujejo `attr_accessor`.
Nato se bomo posvetili modulom in metodam, s katerimi module vključujemo v razrede. Ogledali si bomo, kako po vključenem modulu v razredu uporabljamo metode tega modula. Razdelek bomo zaključili z opisom metod `inherited` in `super`. Za večino primerov bomo podali tudi primere z enakim učinkom v jeziku Python.
Razdelek je razdeljen na tri logične sklope:

- *Razredi in metode*

  V tem razdelku bomo opisali, kako definiramo nov razred. Predstavili bomo razliko med

metodo razreda in metodo objekta razreda. Na koncu razdelka bomo opisali še posebne metode, ki se imenujejo `attr_accessor`-ji.

- *Moduli*

V razdelku bomo najprej povedali kaj moduli so, kako jih definiramo in zakaj jih uporabljamo. Nato bomo opisali metode, s katerimi module vključujemo v razrede.

- *Uporaba metode inherited*

V razdelku si bomo ogledali metodo razreda s posebnim pomenom. Metoda se imenuje `inherited`. Če nek razred vsebuje metodo `inherited`, se bo ta izvedla ob vsakem dedovanju iz tega razreda. Dovolj je, da v kodi definiramo razred, ki deduje iz nekega osnovnega razreda z metodo `inherited`. Takoj, ko bo novi razred definiran v pomilniku procesa tekoče aplikacije jezika Ruby, se bo izvedla metoda `inherited` osnovnega razreda. S ključno besedo `super` pa v metodi dedovanega razreda kličemo istoimensko metodo osnovnega razreda.

2.7.1 Razredi in metode

Najprej si poglejmo, kako v jeziku Ruby definiramo nov razred.
V datoteko `test_razred.rb` zapišemo naslednjo kodo:

```
class Razred

  def initialize()

    puts "Nov objekt"

  end

end
```

S tem smo definirali razred z imenom `Razred`. Dodali smo mu tudi konstruktor `initialize`. To je metoda, ki se pokliče, ko kreiramo nov objekt. Nov objekt ustvarimo z ukazom:

```
ruby> objekt = Razred.new

Nov objekt
```

Vidimo, da se je na standardni izhod izpisala vrstica `Nov objekt`, kar pomeni, da se je ob klicu metode razreda `new` izvedel konstruktor `initialize`. V Pythonu enak učinek dosežemo z naslednjo kodo, ki jo zapišemo v datoteko `test_razred.py`:

```
class Razred:

   def __init__(self):

      print ("Nov objekt")
```

Tudi tu smo definirali konstruktor z imenom `__init__`, v katerem bomo izpisali niz `"Nov objekt"`. Nov objekt pa ustvarimo z ukazom:

```
python> objekt = Razred()

Nov objekt
```

Podobno kot konstrukor definiramo v obeh jezikih metode objektov. Dodajmo v razred `Razred` naslednjo kodo:

```
def metoda(param)

   @a = param

   puts "metoda: spremenljivki a smo nastavili vrednost #{a.to_s}"

end
```

Metodo `metoda` pokličemo na ustvarjenem objektu:

```
ruby> objekt.metoda(5)
```

```
metoda: spremenljivki a smo nastavili vrednost 5
```

V Pythonu isto dosežemo z:

```
def metoda(self,param):
    self.a = param
    print("metoda: spremenljivki a smo določili vrednost %d" %self.a)
```

Metodo `metoda` pokličemo na ustvarjenem objektu:

```
python> objekt.metoda(5)
metoda: spremenljivki a smo določili vrednost 5
```

Opazimo lahko, da sta načina definicij metod objekta zelo podobna. V obeh metodah smo predstavili tudi uporabo spremenljivk, ki pripadajo objektu razreda. V Rubyju take spremenljivke označimo z znakom `@`. Tako smo v metodi `metoda` definirali spremenljivko `@a`. V Pythonu pa do spremenljivke objektov dostopamo preko ključne besede `self`.
Sedaj bomo predstavili še metode in spremenljivke, ki pripadajo razredu in ne objektom razreda. V Rubyju dodamo v razred `Razred` naslednjo kodo:

```
class Razred
  @@cc = 0
  def self.metoda_razreda(param)
    @@cc = param
  end
end
```

Kot vidimo, te metode označimo tako, da pred njihovo ime napišemo `self`. Spremenljivke, ki pripadajo razredu in ne posameznemu objektu pa označimo s @@.
V Pythonu take spremenljivke in metode imenujemo statične spremenljivke in statične metode. Statično metodo tam določimo z uporabo ključne besede `@staticmethod`:

```
class Razred:
   cc = 0
   @staticmethod
   def metoda_razreda(param):
      Razred.cc = param
```

Metode razreda izvajamo na razredu. Ni nam potrebno ustvarjati objektov razreda:

```
python & ruby> Razred.metoda_razreda(5)
> Razred.cc
5
```

Sedaj bomo predstavili še nekaj metod jezika Ruby s posebnim pomenom. Če sledimo konceptu objektnega programiranja, moramo za izvedbo katerekoli akcije v programu uporabiti metodo nekega objekta ali razreda. To velja tudi za dostop do spremenljivk objekta. V Rubyju z ključno besedo `attr_accessor` definiramo dostop do spremenljivk tega razreda ali objektov tega razreda. Sintakso si bomo ogledali na primeru:

```
class Razred
  attr_accessor :a
end
```

S tem, ko smo navedli ključno besedo `attr_accessor` skupaj s simbolom `:a`, smo v bistvu implicitno definirali dve metodi, s katerimi bomo dostopali do spremenljivke `a`. Metodo

```
def a=(param)
```

```
    @a = param
end
```

bomo uporabljali za prirejanje vrednosti spremenljivke. Metodo

```
def a
  @a
end
```

pa bomo uporabljali, da dobimo vrednosti spremenljivke:

```
ruby> objekt.a = 3
Spremenljivki a smo priredili vrednost
=> 3
> objekt.a
Iz spremenljivke a smo odčitali vrednost
=> 3
```

Če želimo le dostopati do vrednosti spremenljivke in je ne bomo spreminjali, uporabimo ključno besedo `attr_reader`. Če bi želeli le spreminjati vrednosti spremenljivke, pa bi uporabili ključno besedo `attr_writer`.
Vgrajene metode `attr_accessor`, `attr_reader` in `attr_writer` nam torej omogočajo preprosto prirejanje in vračanje vrednosti spremenljivk objekta nekega razreda. Če pa želimo na primer omejiti vnose vrednosti spremenljvk le na določene vrednosti, moramo metodo `a=` ustrezno spremeniti. Oglejmo si primer definicije metode `a=` v razredu `Razred`, s katero bomo preprečili vnašanje negativnih števil v spremenljivko `a`:

```
def a=(param)
   if param >= 0
      @a = param
   else
       raise "vrednost spremenljivke a ne more biti negativna"
   end
end
```

S tem smo preprečili vnose negativnih števil v spremenljivko `a`:

```
ruby> objekt = Razred.new
=> #<Razred:0xb76de950>
ruby> objekt.a = -1
RuntimeError: vrednost spremenljivke a ne more biti negativna
       from (irb):28:in a='
       from (irb):33
       from :0
```

Za zaključek razdelka si bomo ogledali še dedovanje. Pri Rubyju lahko razred deduje le iz enega razreda, dedovanje iz večih razredov ni mogoče.

```
class Osnova
  def metoda
     puts "metoda osnova"
  end
end

class Deduje < Osnova
end
```

Če ustvarimo objekt razreda `Deduje`, na njem torej kličemo tudi metode razreda `Osnova`.

```
ruby> d = Deduje.new
```

```
=> #<Deduje:0xb783531c>
> d.metoda

metoda osnova
```

Python v nasprotju z Ruby podpira tudi dedovanje iz večih razredov:

```
class Osnova1:

  def metoda1(self):

    print("metoda - Osnova1")

class Osnova2:

  def metoda2(self):

    print("metoda - Osnova2")

class Deduje(Osnova1,Osnova2):

  """razred Deduje"""
```

V objektu razreda `Deduje` lahko kličemo metode vseh razredov iz katerih razred deduje:

```
python> d  = testrazred.Deduje()

>>> d.metoda1()

metoda - Osnova1

>>> d.metoda2()

metoda - Osnova2
```

Ker je dedovanje v Rubyju mogoče le iz enega razreda, prihaja pri zahtevnejših programih do vključevanja modulov. Moduli v Rubyju predstavljajo imenske prostore, v katerih so definirane metode, lahko pa tudi celi razredi. Z vključitvijo določenega modula v razred so metode, definirane v tem modulu, avtomatično dostopne znotraj razreda. Metode s katerimi vključujemo module v razrede, si bomo ogledali v naslednjem razdelku.

2.7.2 Moduli

Moduli so imenski prostori v jeziku Ruby. Znotraj imenskih prostorov definiramo posamezne metode imenskega prostora ali cele razrede, ki modulu pripadajo. Nov modul definiramo z uporabo ključne besede `module`:

```
module Moduletest

   ...

end
```

Imena modulov se vedno začnejo z veliko začetnico.
V razrede lahko vključujemo poljubne module. S tem nam metode modulov in razredi, ki so v modulih definirani, postanejo avtomatično dostopni v razredu, v katerega smo posamezni modul vključili.
Prva metoda, ki si jo bomo ogledali, je metoda `require`. Metoda `require` se uporablja pri nalaganju modulov jezika Ruby. V datoteki, ki jo podamo kot parameter, se lahko nahajo deli, ki so ukazi za izvrševanje in deli, ki predstavljajo definicijo modulov. Pri izvedbi ukaza `require` se izvedejo deli datoteke, ki predstavljajo ukaze za izvrševanje. Če je v datoteki definiran tudi določen modul, je po izvedbi definiran tudi v našem programu, seveda znotraj razreda, v katerem smo pognali ukaz `require`.
Oglejmo si primer. Ustvarimo datoteko `moduletest.rb` z naslednjo vsebino:

```
#! /usr/bin/env ruby
puts "Uvožena datoteka ruby moduletest.rb"
```

```
module Moduletest
  class Test
   def testna_metoda
      puts "Razred Moduletest.Test metoda testna_metoda"
   end
  end
end
```

Nato ustvarimo datoteko `importtest.rb`, v kateri bomo modul vključili in ustvarili objekt razreda `Test`, ter na njem poklicali metodo `testna_metoda`:

```
#!/usr/bin/env ruby

require "moduletest.rb"

Moduletest::Test.new.testna_metoda
```

Z uporabo ukazne vrstice poženemo skripto `importtest.rb`:

```
ruby> ./importtest.rb

Uvožena datoteka ruby moduletest.rb

Razred Moduletest.Test metoda testna_metoda
```

Metodi `require` vedno podamo parameter tipa niz, s katerim določimo ime (skupaj s potjo, če je potrebno) izvršilne datoteke. Kadar določamo pot, vedno določimo relativno pot. Slednjo izberemo zato, ker se na različnih sistemih izvršilne datoteke lahko nahajajo na različnih poteh. Začetek poti, ki je odvisen od sistema, dodamo v globalno spremenljivko `$LOAD_PATH`. V tej globalni spremenljivki se hranijo vse poti, na katerih metoda `require` pri izvajanju kode išče izvršilne datoteke.
Ime izvršilne datoteke lahko podamo na dva načina:

- Klic `require "test/datoteka.rb"` pomeni, da smo točno določili ime datoteke z relativno potjo od trenutnega imenika, na katerem se nahaja.
- Ob klicu `require "datoteka"` (torej brez končnice `.rb`) okolje išče datoteko (z dodano končnico `rb`) na vseh poteh, ki so definirane v globalni spremenljivki `$LOAD_PATH`.

Metoda `require` si zapomni vse datoteke, ki jih je že naložila, zato se pri morebitnem ponovnem nalaganju datoteke z metodo `require` ne zgodi nič. Če želimo posamezno datoteko v programu ponovno naložiti, moramo uporabiti metodo `load`.
Od metode `require` se razlikuje po tem, da ji lahko podamo le ime datoteke s končnico `.rb`. Torej ni možno, da bi metoda `load` sama s pomočjo sistemske spremenljivke `$LOAD_PATH` poiskala ustrezno datoteko. Druga pomembna razlika je v tem, da metoda `load`, datoteko naloži še enkrat, tudi če je pred tem že bila naložena. Če so torej v datoteki izvršilni ukazi, se izvedejo vsakič, ko uporabimo `load`.
V Pythonu obstaja metoda, ki se obnaša zelo podobno kot metoda `require`, to je metoda `import`.
Moduli so tudi v Pythonu imenski prostori. V Pythonu module preprosto definiramo z imenom datoteke. Ime datoteke predstavlja ime modula, v datoteki sami pa ne uporabljamo nobenih ključnih besed (pri Rubyju modul definiramo v datoteki s ključno besedo `module`).
V Pythonu definirajmo datoteko `moduletest.py`, ki predstavlja modul `moduletest`:

```
#!/usr/bin/env python3

print("Uvožen python modul moduletest");

class Test:

    def testna_metoda(self):
        print("razred moduletest::Test metoda testna_metoda")
```

Za testiranje bomo uporabili datoteko `importtest.py`, v kateri bomo modul `moduletest` vključili in poklicali testno metoda objekta razreda Test, ki je v modulu definiran:

```
#!/usr/bin/env python3

import moduletest

moduletest.Test().testna_metoda()
```

Ko poženemo skripto `importtest.py`, dobimo:

```
python> ./importtest.py

Uvožen python modul moduletest

razred moduletest::Test metoda testna_metoda
```

V tem primeru se metodi pri obeh jezikih obnašata enako. Poudariti je treba, da pri Pythonu lahko vključimo le dele modula z uporabo ukaza `from`. Prav tako lahko ob vključevanju razrede preimenujemo. Razred `Test` denimo lahko shranimo kot `t` :

```
>>> from moduletest import Test as t

Uvožen python modul moduletest

>>> t().testna_metoda()

razred moduletest::Test metoda testna_metoda
```

Oglejmo si sedaj še dve metodi jezika Ruby. To sta `include` in `extend`. S tema metoda lahko razširimo nabor metod nekega razreda. Vanj lahko vključimo metode nekega drugega modula, z njim pa tudi razrede, ki jih vključeni modul vsebuje. Oglejmo si dva primera vključevanja modulov v razred.

V datoteki `moduletest.rb` najprej definiramo modula `ModuleInclude` in `ModuleExtend`:

```
#!/usr/bin/env ruby

module ModuleInclude

   def test

      puts "ModuleInclude::test"

   end

end

module ModuleExtend

   def test

      puts "ModuleExtend::test"

   end

end
```

V datoteko `test.rb` bomo zapisali kodo, s katero bomo prikazali vključevanje modulov v razrede. Najprej bomo z uporabo metode `require` naložili vsebino datoteke `moduletest.rb`. Nato pa bomo modula `ModuleInclude` in `ModuleExtend` vključili v testna razreda `Test1` in `Test2`. Če modul `ModuleInclude` v razred `Test1` vključimo z uporabo ključne besede `include`, lahko metode vključenega modula uporabljamo povsem enako kot objektne metode razreda `Test1`. Metode modula se v tem primeru torej "obnašajo", kot bi bile definirane v razredu `Test1`. Ko pa smo modul `ModuleExtend` vključili v razred `Test2` z `extend`, so metode modula avtomatično dostopne kot metode razreda `Test2`, torej kot razredne, statične metode. Če torej uporabimo `extend`, se metode "obnašajo", kot bi bile v razredu `Test2` napisane z `self.ime_metode`:

```
#!/usr/bin/env ruby
require "moduletest"


class Test1
  include ModuleInclude
end
Test1.new.test


class Test2
  extend ModuleExtend
end
Test2.test
```

Skripto `test.rb` poženemo z naslednjim ukazom:

```
ruby> ./test.rb
ModuleInclude::test
ModuleExtend::test
```

Kot vidimo, razrede lahko širimo tako, da vanje vključimo metode definirane v modulih. Metodo `include` uporabimo, kadar želimo dodajati objektne metode. Z metodo `extern` pa dodajamo metode razreda.

2.7.3 Uporaba metode inherited

Na primeru si bomo ogledali še delovanje posebne metode razreda `self.inherited`. Kot vemo, ključna beseda `self` v imenu metode pomeni, da gre za metodo razreda in ne objekta razreda. Metodi podamo parameter `base`, ki predstavlja razred, ki deduje. Kadar je v kakem razredu ta metoda definirana, se le-ta izvede takrat, ko je razred s to metodo dedovan. Do tega pride, ko se ob izvajanju poljubne skripte naloži razred, ki deduje iz razreda, ki vsebuje metodo `inherited`. Poglejmo si naslednji primer.
V skripti `test_inherited.rb` imamo definiran modul z dvema razredoma. Prvi je `Osnova`, ki vsebuje metodo razreda `self.inherited`. Drugi razred je razred `Deduje`, ki deduje iz razreda Osnova:

```
module TestInherited
   class Osnova
     def self.inherited(base)
        puts "Osnova: Razred #{base} me je dedoval"
     end
   end
   class Deduje < Osnova
   end
end
```

Poglejmo, kaj se zgodi, ko z ukazom `require "testinherited"` naložimo modul:

```
> require "testinherited.rb"
Osnova: Razred TestInherited::Deduje me je dedoval
```

```
=> true
```

Vidimo, da se je metoda `inherited` izvedla ob tem, ko smo z `require` naložili vsebino datoteke `test_inherited.rb`.
V implementaciji okolja Rails se metoda `inherited` pogosto uporablja. Iz nje se kličejo konstruktorji objektov dedovanih razredov. Pogosto imamo v verigi dedovanja v veliko razredih definirano metodo `self.inherited`. Če na primer definiramo razred `Deduje2`, ki deduje iz razreda `Deduje` z metodo `self.inherited`, ta pa deduje iz osnovnega razreda `Osnova` z metodo `self.inherited`, se bo naprej izvedla metoda `self.inherited` razreda `Osnova`, za njo pa še metoda `self.inherited` razreda `Deduje`.

Druga pomembna metoda, ki je povezana z dedovanjem, je metoda `super`. Oglejmo si primer njene uporabe. S klicem metode `super` v objektni metodi `EnakoIme` razreda `Deduje` kličemo metodo z enakim imenom v razredu `Osnova`, iz katerega deduje razred `Deduje`. Kodo shranimo v datoteko `test_super.rb`:

```
#!/usr/bin/env ruby
class Osnova
  def EnakoIme
    puts "Metoda razreda Osnova z imenom EnakoIme"
  end
end
class Deduje < Osnova
  def EnakoIme
    super
    puts "Metoda razreda Deduje z imenom EnakoIme"
  end
end


NovOb = Deduje.new
NovOb.EnakoIme
```

Definirali smo torej razred `Osnova` in razred `Deduje`, ki deduje iz razreda `Osnova`. V obeh razredih imamo definirano metodo z enakim imenom `EnakoIme`. V metodi razreda `Deduje` izvedemo ukaz `super`, s katerim bomo poklicali istoimensko metodo razreda `Osnova`.
V zadnji vrstici datoteke izvedemo še inicializacijo objekta razreda `Deduje` in na njem pokličemo metodo `EnakoIme`. Skripto poženemo z ukazom:

```
ruby> ./test_super.rb
Metoda razreda Osnova z imenom EnakoIme
Metoda razreda Deduje z imenom EnakoIme
```

Ker smo v objektni metodi `EnakoIme` razreda Deduje najprej poklicali ukaz `super`, se je najprej izvedla objektna metoda `EnakoIme` razreda `Osnova`, iz katerega deduje razred `Deduje`.

S tem smo zaključili spoznavanje modulov, razredov in metod. Razdelek je ključen za razumevanje naslednjih razdelkov, predvsem podpoglavij razdelka Rails.

2.8 Bloki in procedure

V razdelku bomo spoznali stavčna konstrukta jezika Ruby, bloke in procedure.

Blok je neimenovan stavčni konstrukt jezika Ruby. Sestavljajo ga stavki, ki so od ostale kode ločeni z zavitimi oklepaji `{}` ali z uporabo ključnih besed `do ... end`.
Blok predstavlja več kot le skupino ukazov. Bloke podajamo posebnim metodam, ki so napisane za izvajanje ukazov v podanem bloku ukazov. Pri tem je potrebno vedeti, da neimenovan blok ukazov v posebni metodi, ki ji je podan, ni definiran kot parameter, čeprav ga pri klicu metode vedno dodamo za zapisom imena metode (prave parametre metode zapišemo pred blok, za boljšo preglednost kode jih lahko pišemo v oklepajih ()). Za boljše razumevanje si oglejmo primer klica take metode:

```
metoda(param1,param2) {blok ukazov}
```

Za izvajanje ukazov bloka ukazov se v metodi vedno uporabi klic standardne metode `yield`. To je pogoj, ki mora biti izpolnjen, da metodi ob klicu lahko podamo tudi nepoimenovan blok ukazov. Taka metoda bloku ukazov lahko določi tudi bločne parametre, na katerih se izvedejo ukazi v bloku ukazov. Bločni parametri nimajo povezave s pravimi parametri metode. V metodi jih podamo kot parametre metodi `yield` ob klicu te metode:

```
yield(param_bloka_1, param_bloka_2)
```

V bloku samem so ti parametri definirani na začetku, ograjeni z znakoma | | in tako ločeni od ukazov v bloku ukazov. Za boljše razumevanje si oglejmo dve možni obliki splošne definicije bloka ukazov:

```
{ |param_bloka_1,param_bloka_2,...| ukaz1; ukaz2; ukaz3 }
```

ali

```
do |param_bloka_1,param_bloka_2,...|
   ukaz1
   ukaz2
   ukaz3
end
```

Procedura je poimenovan stavčni konstrukt jezika Ruby. Procedura je objekt razreda `Proc`. Procedura je sestavljena povsem enako kot nepoimenovan blok. Vsebuje lahko torej bločne parametre, definirane na začetku med znakoma | | in blok ukazov.
Procedure prav tako kot bloke podajamo posebnim metodam razredov, ki jih znajo izvajati. Za razliko od nepoimenovanih blokov so procedure objekti, ki jih posebnim metodam lahko podajamo kot parametre. Obstaja več načinov definiranja in uporabe procedur. Spoznali jih bomo v tem razdelku.
Za začetek si oglejmo eno najbolj razširjenih uporab bloka v jeziku Ruby. V razdelku *Ruby, Tabele in slovarji* smo spoznali metodo tabele `map`, ki kot parameter sprejme blok ukazov. Denimo, da imamo tabelo števil, ki jih želimo pomnožiti s 3:

```
ruby> tabela = [1,2,3]

=> [1, 2, 3]

> tabela.map{|st| st=st*3 }

=> [3, 6, 9]
```

V bloku `{|st| st=st*3 }` smo definirali bločni parameter `st`, ki predstavlja števila v tabeli. Na vsakem številu v tabeli se je izvedlo množenje in prirejanje nove vrednosti. Metoda `map` je vsako število iz tabele posebej podala bloku ukazov kot bločni parameter `st` in njem izvedla ukaz v bloku `st=st*3`. Dobljeno potrojeno vrednost je nato priredila podanemu številu iz tabele števil.

Kot smo omenili v uvodu, bi blok lahko zapisali tudi drugače:

```
ruby> tabela.map do |st|

*          st=st*3

> end

=> [3, 6, 9]
```

Uporabili smo drugo oblika zapisa bloka. Začetek bloka smo označili z besedico `do`, konec pa z besedico `end`. V našem primeru je rezultat enak. Vendar pa obstaja pomembna razlika. Zaviti oklepaji `{}` vežejo močneje kot uporaba `do ... end`. To pride do izraza, kadar več metod hkrati kličemo z uporabo bloka. Denimo, da definiramo še metodo, ki bo v naši tabeli števil vsako število povečala za eno:

```
ruby> def povTab(t)
> for i in 0...t.size
>    t[i] = t[i]+1
> end
> return t
> end
```

Če sedaj metodo kombiniramo z zgornjim primerom bloka, opazimo v rezultatu bistveno razliko:

```
ruby>  povTab tabela.map do |st| st = st*3 end
=> [2, 3, 4]


>  povTab tabela.map {|st| st = st*3}
=> [4, 7, 10]
```

V prvem primeru se je metoda `povTab` izvedla na začetni tabeli in vrnila rezultat. Blok `do...end` sploh ni bil upoštevan. V drugem primeru pa blok `{}` veže močneje in se naprej izvrši množenje in šele nato na rezultatu množenja izvede povečanje. Iz tega primera potegnemo pomemben zaključek. Z uporabo oklepajev v stavku

```
ruby>  povTab(tabela).map do |st| st = st*3 end
=> [4, 7, 10]
```

natančno določimo vrstni red izvajanja ukazov. Na ta način postane koda tudi bolj razumljiva. Kadar v bloku z `{}` izvedemo več stavkov, jih med seboj ločimo s podpičjem:

```
> test{("a".."z").map { |c| print c + " "; print "- "  }; puts "konec"}
a - b - c - d - e - f - g - h - i - j - k - l - m - n - o - p - q - r -
s - t - u - v - w - x - y - z - konec
=> nil
```

Na primeru vidimo, kako uporabimo podpičja. Opazimo pa še eno stvar. Bloke lahko tudi gnezdimo. Znotraj zunanjega bloka smo na tabeli črk poklicali metodo `map` in ji podali blok za izpis znakov.

Kot smo povedali že v uvodu, lahko blokom podajamo tudi parametre. Do sedaj smo vedno to tudi storili. Parametri so vedno definirani na začetku bloka in ločeni od stavkov bloka. Parametre ogradimo z začetnim in končnim znakom `|`. Kadar je parametrov več, so znotraj parametrskega dela bloka `| |` med seboj ločeni z vejico. Metoda, ki jo bomo poklicali z blokom ukazov, bo bloku podala bločne parametre ob klicu metode `yield`.

Pripravimo si naslednji razred za seštevanje števil `Test2`:

```
class Test2
   attr_accessor :spr1
   attr_accessor : spr2
   def initialize(param1,param2)
     @spr1 = param1
     @spr2 = param2
   end
```

```
    def test
        yield(spr1, spr2)
    end
end
```

Vemo, da se metode tipa `attr_accessor` uporabljajo za branje in določanje vrednosti spremenljivk objekta razreda. V našem primeru imamo spremenljivki `spr1` in `spr2`, ki določata dve števili. Ti števili podamo kot parametra metodi `initialize`, ki se izvede ob konstrukciji objekta razreda `Test2`. Razred pa definira še za nas trenutno najbolj zanimivo metodo `test`. Znotraj nje se pokliče metoda `yield`. Metoda `yield`, kot vemo iz uvoda, izvede ukaze v bloku ukazov. Najprej poda bloku spremenljivki `spr1` in `spr2` kot bločna parametra. Nato izvrši ukaz v bloku ukazov, torej sešteje vrednosti spremenljivk. Poglejmo, kako izvedemo metodo `test`:

```
ruby> t2 = Test2.new(1,2)
=> #<Test2:0xb77f576c @spr2=2, @spr1=1>
ruby> t2.test{|p1,p2| p1+p2}
=> 3
```

Na primeru vidimo, kako smo v bloku definirali parametra `p1` in `p2` in ju nato z ukazom v bloku sešteli. Metoda `yield`, ki je izvršila bločni ukaz, mu je podala vrednosti lokalnih spremenljivk, ki smo jih prej določili s konstruktorjem.
Kot vidimo, je že za implementacijo preproste metode, ki zna izvajati kodo v bloku, potrebno precej truda. V naslednjih razdelkih bomo večinoma uporabljali razrede, ki imajo take metode že implementirane. Mi jim bomo le podajali bloke ukazov.

Nekoliko lažje je uporabljati poimenovane ukazne bloke. Ti bloki se, kot vemo iz uvoda, imenujejo procedure. Procedure so v bistvu običajni bloki ukazov, katerih naslove pa lahko shranimo in kasneje po potrebi pokličemo.
Kot smo povedali že v uvodu razdelka, so procedure objekti razreda `Proc`. Ogledali si bomo nekaj primerov kreiranja procedur:

```
mproc = Proc.new{ puts "To je navaden blok"}
```

Ustvarili smo novo proceduro z imenom `mproc`. V bloku procedure se izpiše stavek `"To je navaden blok"`. Lahko bi namesto neposrednega kreiranja novega objekta razreda `Proc` uporabili metodo modula `Kernel`, imenovano `lambda`:

```
mproc2 = lambda{ puts "To je drugi navaden blok"}
```

Pri tako preprostih primerih med obema načinoma kreiranja procedur ni nobene razlike. Če pa v bloku uporabimo parametre, nam način z uporabo `lambda` omogoči preverjanje pravilnega števila podanih parametrov bloku. Oglejmo si primer. Uporabili bomo enak "problem" kot prej, ko smo z blokom seštevali števila. Tu bomo uporabili metodo `call` objekta procedure, ki je pri navadnih blokih nismo imeli. Procedure namreč lahko kličemo, medtem ko lahko bloke izvajamo le v metodah, ki to omogočajo:

```
ruby> mproc = lambda{ |p1,p2| p1+p2}
=> #<Proc:0xb77d0368@(irb):9>
ruby> mproc.call(1,2)
=> 3
```

Če proceduro pokličemo z napačnim številom parametrov

```
ruby> mproc.call(1,2,3)
ArgumentError: wrong number of arguments (3 for 2)
```

dobimo izjemo pri izvajanju procedure.
Preverimo še kaj se zgodi, če bi proceduro ustvarili s kreiranjem objekta razreda `Proc`:

```
ruby> mproc = Proc.new{ |p1,p2| p1+p2}
=> #<Proc:0xb77c4e78@(irb):12>
ruby> mproc.call(1,2,3)
=> 3
```

Vidimo, da taka procedura odvečne parametre ignorira.
Poglejmo si še, kako procedure kličemo v metodah objektov razredov. V ta namen napišimo poseben testni razred:

```
class Test
  def test(proc, p2,p3)
    proc.call(p2,p3)
  end
end
```

Parametre, ki jih bo metoda `test` podala bloku, bomo podali kar metodi. Nato bomo v metodi poklicali proceduro `mproc`, ki jo prav tako podamo metodi kot parameter:

```
> mproc = Proc.new{ |p1,p2| p1+p2}
> t = Test.new
> t.test(mproc,3,4)
=> 7
```

Vidimo, da imamo s procedurami precej manj težav kot z neimenovanimi bloki. Dejansko pa lahko tudi neimenovane bloke uporabljamo kot procedure. V ta namen si pripravimo novo testno metodo:

```
def test(p1,p2,&f)
   f.call(p1,p2)
end
```

Z znakom `&` smo neimenovani blok "prevedli" v proceduro. Ko bomo metodi podali blok ukazov kot parameter, bomo z uporabo `&` v parametru `f` shranili naslov procedure. V metodi nato lahko s tega naslova pokličemo metodo `call` in procedura se bo izvedla.

```
ruby> test(1,2){|a,b| a+b}
=> 3
```

Pri klicu opazimo pomembno podrobnost. Blok se podaja izven okroglih oklepajev za ostale parametre. V Rubyju metode sicer lahko kličemo s parametri brez uporabe okroglih oklepajev. V primeru bloka pa bi uporaba okroglih oklepajev okoli bloka povzročila napako v skladnji programa.

Na primerih smo si ogledali lastnosti in uporabo blokov in procedur. Videli smo, da delujejo kot bloki ukazov, ki se izvedejo v povezavi z izvajanjem metode nekega modula ali razreda. V okolju Rails se v kodi bloki in procedure dostikrat uporabljajo. V njih vnešeni ukazi lahko nalagajo in izvršujejo skripte, ter inicializirajo in poganjajo kompleksne objekte, med njimi tudi objekt spletnega strežnika naše aplikacije.

S tem smo razdelek o blokih in procedurah zaključili. V naslednjem razdelku si bomo ogledali še en poseben konstrukt jezika Ruby – Simbole.

2.9 Simboli

Simbol je neke vrste konstanten niz. Uporabljamo jih, če v programu potrebujemo nespremenljiv niz.

Simbol je objekt razreda `Symbol`. Nov simbol definiramo z uporabo podpičja in imenom simbola:

```
:nov-simbol
```

Objekti razreda `Symbol` so nespremenljivi in jim ne moremo prirejati vrednosti. To pomeni, da vanje nikoli ničesar ne zapišemo. Imajo dve predstavitvi – predstavitev kot niz, ki je enaka kar imenu simbola (torej brez :), v našem primeru `"nov-simbol"` in predstavitev v obliki števila (povezana z naslovom v pomnilniku, kjer simbol je). Vsak simbol je torej enolično določen s svojim imenom. Zato `:nov-simbol` vedno predstavlja isti objekt v pomnilniku. Ko je simbol enkrat definiran, ostane v pomnilniku do konca izvajanja programa. V okolju Rails so simboli velikokrat uporabljeni kot parametri pri klicih metod. Uporabljajo se tudi kot ključi članov v objektu tipa slovar (glej razdelek *Ruby, Tabele in slovarji*).
V razdelku si bomo ogledali lastnosti in primere uporabe simbolov. Začeli bomo z opisom razreda `Symbol`, nadaljevali pa s primeri, ob katerih bomo spoznali večino lastnosti in prednosti simbolov.

Za začetek si oglejmo razred `Symbol`. Uporabili bomo znanje, ki smo ga pridobili v razdelku *Ruby, Lastnosti*. Kot smo izvedeli, vsebuje jezik Ruby ukaze, s katerimi lahko izvemo vse informacije o dedovanju razreda, njegovih metodah ter metodah njegovih objektov.
Na razredu `Symbol` v ukazni vrstici jezika Ruby izvedemo naslednja ukaza:

```
ruby> Symbol.ancestors

=> [Symbol, Object, Kernel]

ruby> Symbol.ancestors - Symbol.included_modules

=> [Symbol, Object]
```

Ugotovimo, da razred `Symbol` deduje le iz razreda `Object`, osnovnega razreda jezika Ruby. Z razredom `Object` je vanj vključen tudi modul `Kernel`. Ker nas zanimajo metode razreda, izvedemo še:

```
ruby> Symbol.methods - Object.methods

=> ["all_symbols"]
```

Edina nepodedovana metoda razreda `Symbol` je metoda `all_symbols`. Metoda vrača vse simbole definirane v jeziku Ruby. Če jo izvedemo, ugotovimo, da je v jeziku Ruby definiranih veliko simbolov:

```
ruby> Symbol.all_symbols

[:RUBY_PATCHLEVEL, :vi_editing_mode, :Separator, :TkLSHFT, :one?,
:setuid?, :auto_indent_mode, :setregid, :back, :Fail, :RET, :ELNRNG,
:member?, :TkOp, :AP_NAME, :readbyte, :suspend_context, :oct, :store,
:WNOHANG, :@seek, :autoload, :rest, :IN_INPUT, :close_read, :type,
:filename_quote_characters=, :priority, :getgm, :PI,
:grant_privilege, :obj, :allow_e, :@irb_path,
:basic_word_break_characters, :inspect_mode=, :rcgen, :end_with?,
:__exit__, :Tms, :executable_real?, :Exception2MessageMapper,
:identify_identifier, :set_last_value, :ENOTSOCK, :group_by,
:math_mode=, :throw, ….
```

Nekatere od teh simbolov bomo spoznali v naslednjih poglavjih.
Oglejmo si še metode objekta razreda `Symbol`:

```
ruby> Symbol.instance_methods - Object.instance_methods

=> ["to_sym", "to_proc", "to_i", "to_int", "id2name"]
```

Na voljo imamo pet metod objekta razreda `Symbol`. Med njimi največkrat uporabljamo metodo `to_proc`, pa še to implicitno. Oglejmo si jo kar na primeru. Denimo, da imamo tabelo nizov. Želimo jih spremeniti tako, da se vsi začno z veliko začetnico. To bi lahko naredili z uporabo metode `map` z blokom ukazov (glej razdelek *Ruby, Bloki in procedure*) takole:

```
%w{ prvi drugi tretji }.map { |elt| elt.capitalize! }
```

Lahko pa uporabimo tudi drug način z uporabo simbola:

```
%w{ prvi drugi tretji}.map (&:capitalize)
```

Pri tem smo definirali in uporabili simbol `:capitalize`. Objekti `"prvi"`, `"drugi"` in `"tretji"` so objekti razreda `String`, ta pa vsebuje metodo `capitalize`. Na vsakem objektu se z uporabo metode `Symbol.to_proc` pokliče metodo z imenom, ki je enaka imenu simbola – `capitalize`. Poklicali smo torej metodo `String::capitalize`.
Nekateri pravijo, da je drugi način z uporabo simbola nekoliko lažje berljiv. Drugih prednosti te metode pa v bistvu ni. Ker se ime metode `capitalize` ne spreminja, jo lahko s pomočjo simbola poiščemo in z metodo `to_proc` na objektu tudi pokličemo.
Poglejmo si še nekaj primerov uporabe simbolov. Vsakič, ko kreiramo niz z istim imenom, se v pomnilniku ustvari nov objekt razreda `String`. To zlahka preverimo z uporabo metode osnovnega razreda `Object`, imenovane `object_id`. Ta metoda vrne unikatno število, ki označuje objekt.

```
ruby> a = "test"

=> "test"

ruby> b = "test"

=> "test"

ruby> a.object_id

=> -608984388

ruby> b.object_id

=> -608988208
```

Pri simbolih je zadeva drugačna. Za eno ime obstaja le en simbol. Zato predstavlja oba niza iz zgornjega primera isti simbol z enako identifikacijsko številko objekta:

```
ruby> a.to_sym.id2name

=> "test"

ruby> b.to_sym.id2name

=> "test

ruby> a.to_sym.object_id

=> 87218

ruby> b.to_sym.object_id

=> 87218
```

Ta simbol lahko podamo kot parameter metodam ostalih razredov, ki bodo uporabljale niz z vsebino `"test"`:

```
def metoda(simbol)

   puts simbol.to_s

end

ruby> metoda(:test)

test
```

S tem smo spoznali poglavitni razlog uporabe simbolov. Simboli nam zmanjšujejo porabo pomnilnika v procesu aplikacije.

Simboli so idealni tudi za definiranje ključev slovarjev in dodajanje elementov v slovarje:

```
slovar[:simbolZaPrviKljuč] = "prviPod"
slovar[:simbolZaDrugiKljuč] = "drugiPod"
```

ali

```
slovar  = {  : simbolZaPrviKljuč => "prviPod", : simbolZaDrugiKljuč =>
"drugiPod" }
```

V okolju Rails se kot parametri metod dostikrat uporabljajo slovarji. Ključi slovarjev so v takih primerih večinoma simboli. Ker predstavljajo nespremenljiva imena spremenljivk, je uporaba simbolov v tem primeru smiselna. Kot primer navedimo uporabo simbola `:id` , ki predstavlja identifikacijsko številko razdelka v tabeli razdelkov. Simbol `:id` se uporablja kot ključ v slovarju `params`, ki ga uporabljamo kot slovar parametrov v akcijah krmilnika (glej razdelek *Rails, Krmilnik*) . Simbol `:id` torej predstavlja niz `"id"` s konstantnim naslovom.

V zgornjih primerih smo prikazali lastnosti in tipično uporabo simbolov v jeziku Ruby. S tem smo spoznavanje simbolov zaključili. Veliko primerov uporabe simbolov pa bomo videli v naslednjih razdelkih. Takrat nam bo dobro poznavanje lastnosti in metod simbolov prišlo zelo prav.

2.10 Izjeme

Izjeme so dogodki, ki se zgodijo ob napakah pri izvajanju programa. Ob izjemah se lahko izvajanje programa zaključi, lahko pa, tako kot pri Pythonu, izjeme ulovimo.
V Rubyju so izjeme implementirane kot razredi, ki dedujejo iz osnovnega razreda izjem, imenovanega `Exception`. Obstaja 30 že vnaprej definiranih tovrstnih razredov. Vsak od teh razredov predstavlja poseben tip napak v programu. Tako na primer razred `NoMethodError` omogoča tvorjenje izjeme, ki se zgodi, kadar pokličemo metodo, ki ne obstaja.
Izjeme se torej dogajajo ob napakah pri izvajanju programa. Ruby ob vsaki napaki pregleda sklad programa in poišče najprimernejši mehanizem za obravnavo te izjeme. V razdelku si bomo najprej ogledali, kako se sproži izjema ob napaki v programu. Nato bomo implementirali svoj razred izjem, ki bo dedoval iz razreda `Exception`. Uporabili ga bomo na primeru. Ogledali si bomo ukaz `raise`, s katerim sprožimo izjemo in ukaz `rescue`, s katerim lahko izjemo ulovimo in spremenimo obnašanje programa ob izjemi. Za večino primerov bomo našli sorodne primere v jeziku Python.

Oglejmo si preprost primer napake v programu. Uporabili bomo deljenje z nič, ki sproži izjemo:

```
ruby> puts 10 / 0
ZeroDivisionError: divided by 0
        from (irb):49:in `/'
        from (irb):49
        from C:/Ruby192/bin/irb:12:in `<main>'
```

Napaka v programu je deljenje z nič. Ob tem se je sprožila ena od definiranih izjem in sicer `ZeroDivisionError`. Za imenom izjeme smo dobili še opis napake `"divided by 0"`. Preostale vrstice so izpis sledi sklada programa. V našem primeru, ko imamo le eno vrstico kode, je to precej neuporabno. Precej drugače pa je, ko želimo v večjem programu ugotoviti, v kateri datoteki in kateri vrstici je prišlo do napake.
Oglejmo si sedaj še, kaj se zgodi v podobnem primeru v jeziku Python:

```
python> print( 10/0 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```


Podobno kot pri Rubyju dobimo izjemo `ZeroDivisionError`, ki se sproži ob napaki deljenja z

nič. Tudi tu dobimo izpis sledi sklada programa, le da je pri Pythonu izpis sledi na prvem mestu.

Na primeru si bomo sedaj ogledali lovljenje izjem. Napisali bomo program, v katerem v zanki preberemo 10 naravnih števil in za vsako prebrano število izpišemo recipročno vrednost. V primeru, da smo programu podali število 0 ali nenumerično vrednost, se bo sprožila izjema, ki jo bomo ulovili in izpisali sporočilo uporabniku. Pri tem bomo lovili izjeme razreda `Exception`, iz katerega v jeziku Ruby dedujejo vsi razredi izjem. Pri tem naletimo na zanimiv problem. Celoštevilsko deljenje namreč v Rubyju daje celoštevilske rezultate. To pomeni, da bomo pri vsakem računanju recipročne vrednosti celega števila večjega od 1 dobili rezultat 0. To sicer lahko rešimo s tem, da število pred izračunom pretvorimo v racionalno število. Vendar pri deljenju z racionalnim številom 0.0 ne dobimo izjeme deljenja z nič, ampak rezultat neskončno (`Infinity`):

```
ruby> 1/3
=> 0
> 1/0.0
=> Infinity
```

Ker Ruby omogoča tudi dodajanje metod obstoječim razredom, problem razrešimo tako, da v osnovnemu razredu celih števil, razredu `Fixnum`, dodamo definicijo novega operatorja / .

```
class Fixnum
   def /(num)
     if (num == 0) || ((num.class != Fixnum) && (num.class != Float))
         raise ZeroDivisionError, "Deljenje z nič ali
          nenumeričnim simbolom (ki postane nič po pretvorbi v število)"
     else
         self.to_f/num.to_f
     end
   end
end
```

Torej v primeru, da želimo izračunati recipročno vrednost števila 0 ali nekega objekta, ki ni racionalno ali celo število, vržemo izjemo `ZeroDivisionError`. Enako se Ruby obnaša, če v ukazno vrstico vnesemo denimo ukaza `1/0` ali `1/"aaa"`. Izjemo v Rubyju torej sprožimo z uporabo ključne besede `raise`, ki ji podamo kot parametra tip izjeme in dodatno sporočilo z opisom izjeme.
Z naslednjo kodo izpisujemo recipročne vrednosti prebranih števil:

```
(1..10).each do
    |num|
    puts "Vnesi število: "
    num = gets
    begin
       puts 1/num.chomp.to_f
    rescue Exception => e
      puts "Prišlo je do napake tipa #{e.class}: #{e.message}"
    end
end
```

Kodo, v kateri lahko pride do izjeme, smo vstavili v varovani blok `begin...end`. Za lovljenje pa smo v varovanem bloku uporabili metodo `rescue`. Varovana koda se torej nahaja med ključnima besedama `begin` in `rescue`, obravnavo izjeme pa izvedemo med ključnima besedama `rescue` in `end`. Metodi `rescue` smo podali element slovarja kot parameter. Ključ elementa slovarja je osnovni razred izjem `Exception`, njegova vrednost pa spremenljivka `e`, v katero se bo shranil objekt vržene izjeme. Po lovljenju bomo le izpisali tip in sporočilo vržene izjeme. Oglejmo si delovanje kode iz zgornjih primerov, ki jo shranimo v datoteko `test_izjem.rb`:

```
ruby>./test_izjem.rb

Vnesi število:

1.5

0.666666666666667

Vnesi število:

0

Prišlo je do napake tipa ZeroDivisionError: Deljenje z nič ali
nenumeričnim simbolom (ki postane nič po pretvorbi v število)

Vnesi število:

aaa

Prišlo je do napake tipa ZeroDivisionError: Deljenje z nič ali
nenumeričnim simbolom (ki postane nič po pretvorbi v število)

Vnesi število:

1

1.0

Vnesi število:

2

0.5

Vnesi število:

3

0.333333333333333
```

Program deluje tako, kot smo si zamislili. V Pythonu dosežemo podoben učinek na precej enostavnejši način. Naslednjo kodo shranimo v datoteko `test_izjem.py`:

```
import sys

for i in range(10):

   print("Vnesi število:")

   num = sys.stdin.readline()

   num = num.rstrip('\n')

   try:

      print("%f" %(1/float(num)))

   except Exception as e:

      print("Ujeli smo izjemo %s" %str(e))

end
```

Naložili smo modul `sys`, ki ga bomo uporabili pri branju števil. Prebrano vrstico podobno kot pri Rubyju konvertiramo v število in izračunamo recipročno vrednost. Python avtomatsko vrne

racionalno vrednost izračuna deljenja dveh celih števil:

```
python> ./test_izjem.py
Vnesi število:
1
1.000000
Vnesi število:
2
0.500000
Vnesi število:
1.5
0.666667
Vnesi število:
0
Ujeli smo izjemo float division
Vnesi število:
aaa
Ujeli smo izjemo could not convert string to float: aaa
```

Za konec si oglejmo še, kako napišemo svoj razred izjeme. Na osnovi razreda `Exception` bomo definirali svoj razred in na primeru prikazali proženje in lovljenje izjem. V Rubyju ustvarimo skripto `testexception.rb` z naslednjo vsebino:

```
class TestException < Exception
end
begin
   raise TestException, "Testna izjema sprožena"
rescue TestException => e
   puts e.class.to_s + ": " + e.message
   puts e.backtrace
end
```

Definirali smo nov razred `TestException`, ki deduje iz razreda `Exception`. Ker potrebujemo le lastnosti in metode osnovnega razreda (dejansko smo želeli le "preimenovati" razred `Exception`), dejanske "vsebine" razreda ni. Testno kodo, s katero preverimo delovanje tega razreda, predstavlja blok ukazov `begin...end`. V vrstici `raise TestException, "Testna izjema sprožena"`, sprožimo novo definirano izjemo. V naslednji vrstici `rescue TestException => e` izjemo ujamemo. Ime izjeme smo uporabili kot ključ slovarja, kot njegovo vrednost pa smo določili spremenljivko `e`. V tej spremenljivki z metodo `class` dostopamo do imena izjeme, z metodo `message` pa do sporočila v izjemi. Na koncu pa še izpišemo sled sklada programa z metodo `backtrace`. Če si pogledamo izpis testne skripte `test_exception.rb`

```
ruby>ruby test_exception.rb
TestException: Testna izjema sprožena
test_exception.rb:7:in `<main>'
```


opazimo, da smo dobili izpis zgoraj omenjenih vrednosti razreda `TestException`. V zadnji

vrstici vidimo, da se je izjema sprožila v 7 vrstici skripte `test_exception.rb`.
Pri Pythonu podobno dosežemo s skripto `test_exception.py`:

```
import sys, traceback

class TestException(Exception):

    def __init__(self, value):

       self.value = value

    def __str__(self):

      return repr(self.value)

try:

   raise TestException("Testna izjema")

except TestException as e:

   exc_type, exc_value, exc_traceback = sys.exc_info()

   print(exc_type, " :", exc_value)

   traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
```

Podobno kot pri Rubyju smo definirali nov razred izjeme `TestException`. V konstruktorju tega razreda smo shranili vrednost sporočila izjeme. Dodali smo še funkcijo `__str__`, s katero vračamo vrednost sporočila izjeme v obliki niza znakov. Pri primeru v Rubyju smo uporabili blok `begin rescue end`, v Pythonu pa za isti namen uporabimo blok `try except`. Ukaz za proženje izjem je enak: `raise`. V bloku `except` opazimo, zakaj je bilo potrebno vključiti modula `sys` in `traceback`. Vrednost sporočila izjeme bi sicer lahko izpisali tudi brez uporabe modulov `sys` in `traceback` s klicem metode `e.value`, za izpis tipa izjeme in sledi sklada programa pa potrebujemo podatke, ki jih dobimo z klicem metode `sys.exec_info`. Modul `traceback` in njegovo metodo `print_tb` potrebujemo za izpis sledi sklada programa. Če poženemo zgornjo skripto `testexception.py`, dobimo:

```
python>python test_exception.py

<class '__main__.TestException'>  : 'Testna izjema'

  File "test_exception.py", line 11, in <module>

    raise TestException("Testna izjema")
```

Ta izpis je zelo podoben tistemu zgoraj pri Rubyju. Vsebuje enake informacije o izjemi. Ugotovimo pa lahko, da se je treba pri Pythonu malo bolj potruditi, da dosežemo isti rezultat. To je posledica arhitekture jezika. Pri Rubyju ima večina razredov vključene metode dodatnih modulov. Te metode so tako avtomatično dostopne po kreaciji objekta razreda, ki tak modul vključuje (glej razdelek *Ruby, Razredi, metode in moduli*).

S tem smo zaključili kratek pregled izjem v jeziku Ruby. Ogledali smo si najpomembnejše metode, ki jih uporabljamo za proženje in lovljenje izjem. Videli smo tudi, kako definiramo svoj razred izjeme. Razdelek seveda predstavlja le kratek vpogled v celotno funkcionalnost izjem. Kot smo povedali, v Rubyju obstaja okoli 30 že vgrajenih razredov izjem, iz katerih lahko dedujemo ali pa jih preprosto uporabimo.

3 Rails

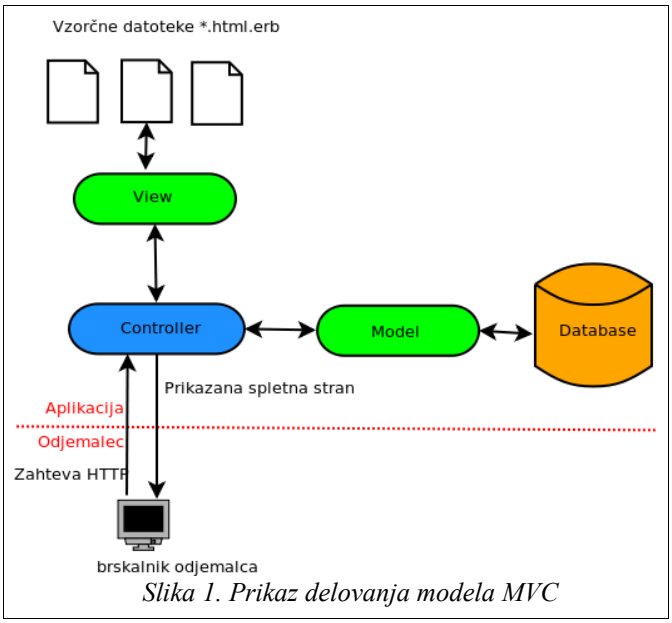
3.1 Zgodovina

Okolje Rails je razvojno okolje za hitro in urejeno razvijanje spletnih aplikacij. Okolje je razvito z uporabo jezika Ruby. Njegov avtor je David Heinemeier Hansson. Prvo različico okolja Rails je

razvil julija 2004 in jo izdal kot odprto kodo. To različico je razvil v sklopu dela na projektu Basecamp. V sklopu projekta Basecamp so razvijali spletno aplikacijo, ki naj bi vsebovala orodja za upravljanje s spletnimi stranmi. V februarju leta 2005 je avtor okolja Rails dovolil shranjevanje sprememb drugih avtorjev v odprto kodo okolja Rails. Oktobra 2007 pa je okolje Rails doživelo razcvet, ko ga je korporacija Apple vključila v novo različico operacijskega sistema Mac OSX, imenovano Leopard (10.5). Leta 2009 je bila izdana pomembna različica okolja Rails 2.3. Ta različica je vsebovala podporo za uporabo vzorčnih predlog, paketa Rack za komunikacijo s spletnim strežnikom in gnezdenje spletnih obrazcev.
Zadnja različica (september 2011) okolja Rails je različica 3.1., v diplomski nalogi pa smo za razvoj aplikacije naše diplome uporabili različico 3.0.6.

3.2 Lastnosti

Okolje Rails uporablja za razvoj spletnih aplikacij arhitekturo Model View Controler (MVC). Spletne aplikacije napisane v skladu z arhitekturo MVC so narejene iz treh sestavnih delov. Krmilnik (Controller (C)) predstavlja možgane aplikacije. Krmilnik skrbi za obravnavo zahtev HTTP. Pridobiva podatke od Modela (M) aplikacije in jih posreduje prikazovalniku (View (V)). Model aplikacije (M) je odgovoren za delo z bazo podatkov aplikacije. Prikazovalnik je odgovoren za generiranje spletnih strani v jeziku HTML. Te strani krmilnik kot odgovore HTTP pošilja nazaj odjemalcu.
Ta vzorec delovanja najbolje predstavi sledeča slika, ki ponazori obravnavo zahteve HTTP v okolju Rails na spletnem strežniku:



Slika 1. Prikaz delovanja modela MVC

V okolju Rails se pri razvoju aplikacij uporablja nekaj pomembnih pravil. Prvo pravilo je Dogovor pred nastavitvami (Convention over Configuration (CoC)). V praksi to pomeni, da je v okolju Rails poimenovanje objektov bolj pomembno od nastavitev v konfiguracijskih skriptah. Obstajajo namreč določena pravila poimenovanja, denimo ime razreda modela aplikacije je vedno definirano v ednini. Primer tega vidimo v modelu naše aplikacije, kjer je model poimenovan `Paragraph`. Razred krmilnika aplikacije, ki deluje z modelom `Paragraph`, pa se imenuje `Paragraphs`, torej v množini. Tako sta model in krmilnik imensko povezana, in s tem odpade potreba po posebnih nastavitvah, ki bi povezovale model in krmilnik. Na podoben način sta povezana tudi Krmilnik in Prikazovalnik. Podrobnosti bomo spoznali v naslednjih razdelkih.

Drugo pomembno pravilo se imenuje Ne ponavljaj se (Dont Repeat Yourself (DRY)). Nanaša se na splošno pravilo programiranja, ki prepoveduje ponavljanje enake kode v različnih datotekah iste aplikacije. Okolje Rails vsebuje rešitev z uporabo skupnih datotek. V te datoteke preprosto napišemo kodo, ki se ponavlja in jo potem kličemo v ostalih datotekah.

Značilnost okolja Rails je tudi uporaba generatorjev za generiranje aplikacij in tudi posameznih delov aplikacij. Generatorje bomo podrobneje spoznali v naslednjih razdelkih. Omogočajo nam hiter razvoj spletne aplikacije. Z generatorji namreč ustvarimo aplikacijo, razrede krmilnika, modela in prikazovalnika. Pri razvoju aplikacije z njimi ustvarimo večino kode. Preostala koda, ki jo moramo napisati, predstavlja manjše prilagoditve, s katerimi aplikaciji določimo videz spletnih strani in potek delovanja.

3.3 Generiranje nove aplikacije okolja Rails

Razdelek opisuje proces generiranja nove aplikacije, razvite s pomočjo okolja Rails. Razdeljen je na tri podrazdelke:

- *Kako ustvariti novo aplikacijo okolja Rails*

Če želimo ustvariti novo aplikacijo, ki bo delovala s pomočjo okolja Rails, najprej zgradimo ogrodje te aplikacije. Pri tem si pomagamo z ustreznim ukazom. Ta ustvari ustrezno imeniško strukturo in vrsto datotek. Te datoteke so skripte, ki sodelujejo pri zagonu aplikacije, vsebujejo definicije osnovnih razredov ključnih delov aplikacije in konfiguracijske datoteke z nastavitvami okolja aplikacije. V tem razdelku si bomo ogledali podrobnosti o ukazu, s katerim ustvarimo to imeniško strukturo. Z njim izdelamo ogrodje nove aplikacije, v katero potem dodamo ustrezno kodo. Prikazana je tudi slika ustvarjene strukture imenikov in datotek.

- *Analiza kode, ki se izvede pri ukazu rails new <application>*

Ukaz, ki generira ogrodje nove aplikacije, požene program, ki je del okolja Rails. Ta program se imenuje generator aplikacij okolja Rails. Podrobneje si bomo ogledali kodo, ki se ob izvede ob zagonu generatorja. Razdelek vsebuje dva logična sklopa, ki si sledita v procesu generiranja nove aplikacije:

  - *Skupna koda ob izvajanju ukaza rails*

    V tem razdelku bomo na kratko opisali skupno kodo, ki se izvede vsakič, ko poženemo ukaz tipa `Rails`. Posvetili se bomo predvsem skripti `cli.rb`, ki jo običajno najdemo v imeniku `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/`. Razdelek bomo zaključili z opisom metode `Rails::ScriptRailsLoader.exec_script_rails!` . V tej metodi se določi nadaljnji potek izvrševanja ukaza `rails`. Ukaz `rails` namreč uporabljamo tako pri generiranju nove aplikacije, kot tudi pri zagonu aplikacije (glej razdelek *Rails, Zagon aplikacije in spletnega strežnika*). Omenjena metoda prepozna namen ukaza `rails`, ki ga izvedemo, da bi generirali novo aplikacijo. Zaradi tega nam omogoči izvajanje skript za generiranje ogrodja nove aplikacije.

  - *Nalaganje, inicializacija in zagon glavnega generatorja aplikacije Rails*

    V razdelku bomo spoznali glavni generator za generiranje aplikacij v okolju Rails. Razdelek bomo začeli tam, ker smo končali prejšnjega. Nadaljevali bomo torej z opisom skripte `cli.rb`. Ta izvede nalaganje, inicializacijo in zagon glavnega generatorja aplikacij. Ti postopki bodo podrobno opisani.

- *Organizacija in pomen generiranih imenikov in datotek*

Razdelek podaja kratko razlago generiranih imenikov in datotek. Večino generiranih datotek bomo podrobneje obravnavali v naslednjih poglavjih.

3.3.1 Kako ustvariti novo aplikacijo okolja Rails

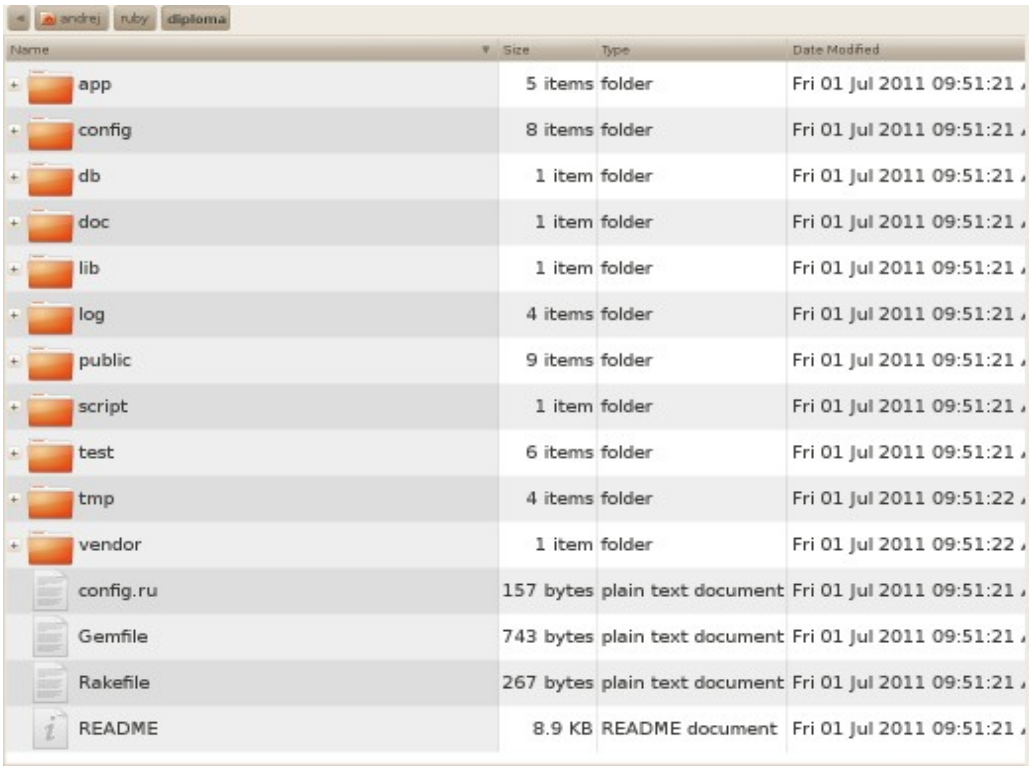
Okostje nove aplikacije Rails ustvarimo z ukazom:

```
rails new <ime_aplikacije>
```

V našem primeru se bo aplikacija imenovala `diploma`. V okno terminala vtipkamo:

```
$ rails new diploma
```

Ukaz ustvari okostje aplikacije. To pomeni, da ustvari sistem imenikov in datotek. Nastane imenik `diploma/` s strukturo podimenikov skupaj s konfiguracijskimi datotekami projekta. Slednje vsebujejo nastavitve, ki določajo način delovanja naše aplikacije. Imenik `diploma/` nastane v imeniku, kjer je ukaz izveden. V našem primeru je to `/home/andrej/ruby/`:



Slika 2. Imeniška struktura aplikacije

3.3.2 Analiza kode, ki se izvede pri ukazu *rails new <application>*

Začeli bomo s skripto, ki se izvede v vsakem primeru, torej vedno, ko izvedemo ukaz `rails`. Skripta `cli.rb` je ključna za nadaljni potek izvrševanja ukazov. V njej si bomo podrobno ogledali in preučili klic metode `Rails::ScriptRailsLoader.exec_script_rails!`. Nato bomo spoznali glavni generator aplikacij okolja Rails. Zagon generatorja bomo preučili ob izvajanju naslednjih skript:

- `application.rb`
- `generators.rb`
- `application.rb`

Prvi dve najdemo v istem imeniku kot `cli.rb`, torej v `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/`, zadnja pa je v podimeniku `generators/`.

3.3.2.1 Skupna koda ob izvajanju ukaza rails

Ne glede na to, kaj sledi ukazu `rails`, se vedno izvede določena koda. Oglejmo si jo.

Ko izvedemo ukaz `rails`, se zažene izvršilna datoteka `/usr/bin/rails`. To je skripta, napisana v jeziku Ruby. V skripti se naloži standardna knjižnica jezika Ruby, imenovana `RubyGems`. Knjižnica je podrobneje opisana v razdelku *Ruby, Paketi (RubyGems)* in se uporablja za namestitev paketov, napisanih v jeziku Ruby. V skripti `rails` se zatem preveri in požene nameščena različica izvršilne datoteke okolja Rails. V našem primeru je to `/usr/lib/ruby/gems/1.8/gems/rails-3.0.6/bin/rails`.
Ta izvršilna datoteka požene skripto za generiranje projekta `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/cli.rb`.
Oglejmo si le dva ukaza iz te skripte:

```
...
require 'rails/script_rails_loader'
...
Rails::ScriptRailsLoader.exec_script_rails!
```

Skripta naloži razred `Rails::ScriptRailsLoader`. Na tem razredu požene metodo razreda `exec_script_rails!`. Ta igra ključno vlogo pri določanju nadaljnjega poteka programa. Oglejmo si jo:

```
def self.exec_script_rails!
    ...
    return unless in_rails_application? ||
                  in_rails_application_subdirectory?
    exec RUBY, SCRIPT_RAILS, *ARGV if in_rails_application?
   ...
```

Poudarjeni pogojni stavek (glej razdelek *Ruby, Pogojni stavki in zanke, še posebej razlago o uporabi skupaj z ukazom return*) pokliče metodi razreda `Rails::ScriptRailsLoader in_rails_aplication?` in `in_rails_application_subdirectory?`. Ti metodi preverita, če je bil ukaz `rails new diploma` pognan v imeniku aplikacije `/home/andrej/ruby/diploma/` ali podimeniku tega imenika. V našem primeru smo ukaz `rails new diploma` pognali v praznem imeniku `/home/andrej/ruby/`. Ker v njem ni skripte `rails/server`, obe metodi vrneta `false`. Pogoj torej ni izpolnjen. Izvede se `return`, in izvajanje metode je zaključeno.
Povsem drugače se metoda obnaša, če je pogoj izpolnjen, kar bomo podrobneje spoznali v razdelku *Rails, Zagon aplikacije in spletnega strežnika*.
V skripti `cli.rb` se z vrstico, ki vsebuje `Rails::ScriptRailsLoader.exec_script_rails!` konča koda, ki je skupna izvrševanju vsakega ukaza, ki ga izvedemo z `rails`. V nadaljevanju skripte `cli.rb` pa so ukazi, ki so specifični za nastanek nove aplikacije (novega okostja).

3.3.2.2 Nalaganje, inicializacija in zagon glavnega generatorja aplikacije Rails

Ukazi, ki so sedaj na vrsti, poskrbijo za nalaganje, inicializacijo in zagon glavnega generatorja za generiranje aplikacij Rails.
Preden začnemo z njihovim opisom, je potrebno povedati nekaj besed o generatorjih okolja Rails. Generatorji okolja Rails so programi okolja Rails, ki jih izvedemo, da z njimi avtomatsko ustvarimo določene dele aplikacije. Z njimi lahko denimo ustvarimo ogrodje aplikacije, nov krmilnik aplikacije, nov model aplikacije ali nov pomožni modul aplikacije. Če potrebujemo nov krmilnik aplikacije, poženemo generator krmilnikov aplikacije. Za nov model aplikacije poženemo generator modelov aplikacije. V tem razdelku bomo spoznali glavni generator aplikacij Rails za generiranje ogrodja aplikacije. Nekatere preostale generatorje okolja Rails

bomo spoznali v naslednjih razdelkih.
V nadaljevanju se bomo sklicevali na vrsto skript, ki se vse nahajajo v imeniku `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/` ali njegovih podimenikih. Če je skripta v podimeniku, bomo pred njenim imenom navedli še ustrezni del poti.
V skripti `cli.rb` (spomnimo se, da smo pri opisu dogajanja v tej skripti z opisom ukaza `Rails::ScriptRailsLoader.exec_script_rails!` zaključili z ukazi, ki se izvedejo vedno, ko poženemo `rails`) je na vrsti ukaz:

```
require 'rails/commands/application'
```

S tem ukazom se naloži vsebina skripte `commands/application.rb`. Kot smo povedali v razdelku *Ruby, Razredi, metode in moduli* se ob nalaganju z ukazom `require` skripta izvede. V tej skripti se izvede nalaganje in zagon glavnega generatorja aplikacij okolja Rails. Oglejmo si le ključni del kode te skripte in sicer ukaza:

```
require 'rails/generators/rails/app/app_generator'

Rails::Generators::AppGenerator.start
```

Prvi ukaz naloži vsebino skripte `generators/rails/app/app_generator.rb`.
V njej se nahaja definicija razreda `Rais::Generators::AppGenerator`. To je razred glavnega generatorja aplikacij okolja Rails. Odgovoren je za postavitev okostja naše aplikacije. Pri tem si pomaga s tem, da se v imeniku `generators/rails/app/` nahaja struktura podimenikov in datotek, ki je praktično identična imenikom in datotekam, ki smo jih dobili po pognanem ukazu `rails new diploma`. To so vzorčne datoteke, s katerimi `Rails::Generators::AppGenerator` ustvari okostje naše aplikacije.

Da bomo lahko razumeli, kaj se dogaja, moramo vedeti, da je razred `Rails::Generator::AppGenerator` nastal z dedovanjem iz razreda `Rails::Generators::Base`, ki spet deduje iz razreda `Thor::Group`.

`Thor` je paket v jeziku Ruby, ki predstavlja zmogljivo okolje za procesiranje programskih opcij in delo z datotekami. V različici 3 okolja Rails so vsi generatorji osnovani na modulih paketa `Thor`. Za več podrobnosti o tem paketu si lahko ogledamo dokumentacijo paketa `Thor` (glej *Literatura in viri*).

Z zadnjim ukazom skripte `rails/application.rb` - `Rails::Generators::AppGenerator.start` ustvarimo objekt generatorja `Rails::Generators::AppGenerator` ter poženemo vse metode tega objekta.
Oglejmo si, kaj se zgodi pri izvajanju metod generatorja. Spomnimo se, da se v imeniku generatorja nahaja vzorčna struktura podimenikov in datotek.
Metode generatorja ločijo na vzorčni strukturi tri glavne tipe podatkov: imenike, navadne datoteke in vzorčne datoteke. Generatorji najprej v imeniku, kjer je skripta pognana, ustvarijo vzorčni identično imeniško strukturo. Navadne datoteke (primer je, denimo datoteka `README`) skopirajo iz vzorčne strukture na pravo mesto v imeniški strukturi novo nastajajoče aplikacije. Vzorčne datoteke poleg ostalega teksta vsebujejo bloke, ki so napisani v jeziku ERB (`Embedded Ruby`). V bloku ERB datoteke so ukazi, ki so napisani v jeziku Ruby. ERB bomo podrobneje spoznali v razdelku *Rails, Prikazovalnik*.
Metode generatorja pri generiranju datotek iz vzorčnih datotek izvršijo bloke ERB vzorčnih datotek. V teh blokih uporabijo ustrezne vrednosti parametrov ukaza `rails new diploma` (v našem primeru je edini parameter ime aplikacije - `diploma`), informacije, pridobljene od sistema (na primer, kje se nahaja izvršilna datoteka `rails`), ter privzete nastavitve ukaza `rails`.
S tem, ko se izvede koda v bloku ERB, se njegova vsebina zamenja z rezultatom ustreznih ukazov jezika Ruby. Na ta način iz vzorčne datoteke dobimo datoteko naše aplikacije s pravimi podatki. Metoda generatorja jo lahko shrani na pravo mesto v imeniški strukturi naše aplikacije.
Pomen kreiranih datotek bomo spoznali v naslednjem razdelku. Na kratko si v zaporedju

njihovega izvajanja oglejmo še metode glavnega generatorja.
Prva metoda generatorja, ki se izvede, je `create_root`. Metoda ustvari imenik naše aplikacije `/home/andrej/ruby/diploma/`. Za njo se izvede metoda `create_root_files`, s katero se v imeniku naše aplikacije generirajo datoteke `config.ru`, `Rakefile`, `.gitignore` in `Gemfile`. Metoda `create_app_files` ustvari glavne datoteke modela, prikazovalnika in krmilnika aplikacije. Metoda `create_config_files` ustvari datoteko za usmerjanje `config/routes.rb`, ter datoteki, ki se uporabljata ob zagonu aplikacije: `config/environment.rb` in `config/application.rb`. Metoda `create_boot_file` ustvari datoteko `config/boot.rb`, ki se tudi uporablja pri zagonu aplikacije. Z metodama `create_active_record_files` in `create_db_files` se ustvarita datoteka s shemo baze podatkov, ter konfiguracijska datoteka baze podatkov.
Sledi še generiranje dokumentacije, datotek za hranjenje sporočil aplikacije, javnega imenika (`public/`), datotek kaskadnih slogovnih predlog CSS, skript jezika Javascript, datoteke `script/rails`, začasnega imenika in imenika za pakete drugih aplikacij.
S tem je generiranje imenikov in datotek za okostje naše aplikacije `diploma` končano. V naslednjem razdelku nas čaka kratek pregled tega, čemu služijo posamezni imeniki in datoteke.

3.3.3 Organizacija in pomen generiranih imenikov in datotek

Ena glavnih prednosti okolja Rails je enovita struktura imenikov pri vseh aplikacijah okolja Rails. To pripomore k večji berljivosti in boljši organiziranosti kode. Iz slike 2 (na strani 50) je razvidno, da je ukaz `rails new diploma` ustvaril imenik `diploma/`, na njem pa kompleksno strukturo podimenikov in generiranih datotek. Oglejmo si njene najbolj pomembne člene:

- `app/` V imeniku `app/` se bodo hranile vse najpomembnejše datoteke - jedro aplikacije. Generator aplikacije je na njem ustvaril naslednjo imeniško strukturo:
  - `app/models/` Po generiranju okostja aplikacije je ta imenik prazen. Kasneje bo vseboval datoteke, ki v naši aplikaciji, strukturirani po modelu MVC, predstavljajo model (M). Model MVC je podrobneje opisan v razdelku *Rails, Lastnosti*. V teh datotekah bo koda, ki bo poganjala transakcije na bazi podatkov in vračala podatke krmilniku.
  - `app/views/` Imenik pripada delu modela MVC, ki se imenuje prikazovalnik (View (V)). V njem bomo shranjevali vzorčne datoteke s končnico `html.erb`. To so vzorčne datoteke, iz katerih se med delovanjem spletne aplikacije generirajo spletne strani. Imenik `app/views/` vsebuje podimenik `app/views/layouts/`. Ta je namenjen vzorčnim datotekam, kjer se generirajo tisti deli spletnih strani, ki so skupni vsem spletnim stranem aplikacije. V njem je generator naredil datoteko `application.html.erb`. Ta vzorčna datoteka določa videz vseh spletnih strani. V njej povemo, katere kaskadne slogovne predloge CSS in knjižnice jezika Javascript bomo uporabljali (bodisi tako, da jih navedemo, bodisi da se skličemo na ustrezne datoteke). Več o njej bomo izvedeli v razdelku *Rails, Prikazovalnik*.
  - `app/controllers/` Imenik bo vseboval krmilnike naše aplikacije. Krmilnik (Controller (C)) je prav tako del modela MVC. Krmilnik sestavljajo posebne metode, imenovane akcije. Akcije se prožijo po prejetih zahtevah HTTP. Akcija krmilnika na podlagi podatkov zahteve HTTP od modela aplikacije zahteva ustrezne podatke iz baze podatkov. Te podatke nato posreduje kot odgovor HTTP v obliki spletne strani, generirane iz vzorčnih datotek prikazovanika. Več o krmilniku bomo izvedeli v razdelku *Rails, Krmilnik* . V imeniku se pri generiranju aplikacije generira datoteka `application_controller.rb`. V datoteki je definiran osnovni razred krmilnikov aplikacije `ApplicationController`, iz katerega bodo dedovali vsi krmilniki naše aplikacije.

- `app/helpers/` Imenik bo vseboval pomožne module aplikacije. Ime pomožnega modula je vedno sestavljeno iz imena krmilnika in besede `helper`. V vsakem krmilniku aplikacije je avtomatično vključen njegov prileżni pomožni modul. S tem so metode tega modula dostopne v krmilniku in vzorčnih datotekah prikazovalnika. Metode pomožnega modula se ponavadi kličejo v stavčnih blokih ERB vzorčnih datotek in služijo za urejanje podatkov pred prikazovanjem. V imeniku se pri generiranju aplikacije generira datoteka `application_helper.rb`. Ta vsebuje modul, ki je pomožni modul cele aplikacije. Vključen je v osnovni razred krmilnikov aplikacije, iz katerega dedujejo vsi krmilniki aplikacije.

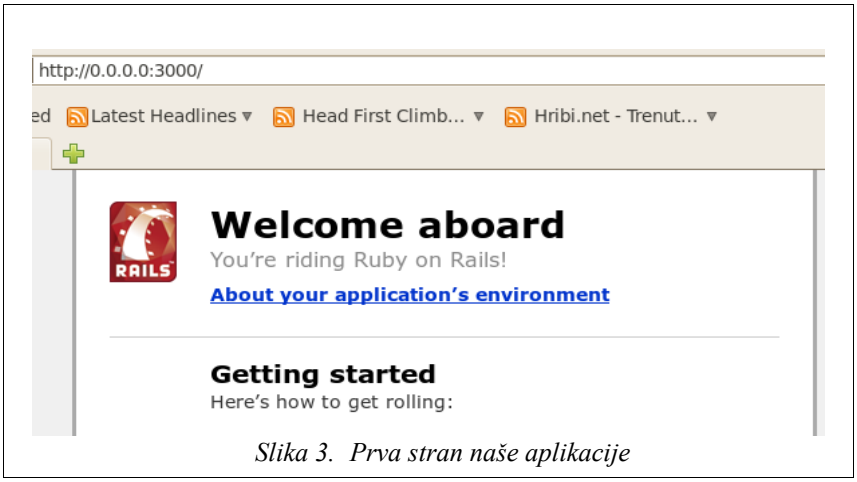
- `config/` Imenik vsebuje konfiguracijske datoteke aplikacije. V njem se ob generiranju aplikacije generirajo skripte, ki se zaženejo pri zagonu aplikacije. To so `application.html.erb`, `boot.rb` in `environment.rb`. Podrobneje so predstavljene v naslednjem razdelku. Na imeniku `config/` se poleg skript za zagon aplikacije generira tudi datoteka `routes.rb`. V njej so definirane vse usmeritve. Usmeritve določajo, kako se zahteve HTTP preslikajo v ustrezne akcije krmilnikov. Več o tem bomo izvedeli v razdelku *Rails, Usmerjanje*. V imeniku `config/` najdemo še eno pomembno datoteko. To je datoteka `database.yml`. Vsebuje konfiguracijo osnovnih podatkov za dostop do baz podatkov aplikacije Rails . Podrobneje jo bomo spoznali v razdelku *Rails, Konfiguracija baze podatkov*.
- `db/` Imenik vsebuje shemo baze podatkov aplikacije in podimenik `migrate/` v katerem se hranijo migracije. Migracije so spremembe, izvedene na bazi podatkov. Več o tem bomo izvedeli v razdelku *Rails, Konfiguracija baze podatkov*.
- `doc/` Imenik vsebuje dokumentacijske datoteke aplikacije, ki se generirajo z ukazom `rake doc:app`.
- `lib/` Imenik vsebuje kodo, ki ni del modela MVC. Ta koda je organizirana v module in razrede, ki so dostopni kjerkoli v aplikaciji, saj je imenik `lib/` vključen v izvršilno pot aplikacije.
- `log/` Imenik vsebuje datoteke `log`, kjer se beležijo vsi izvedeni ukazi. Beležke so zelo pomembne pri analiziranju in odpravljanju napak.
- `public/` Imenik vsebuje slike (`.jpg`, `.png`...), kaskadne slogovne predloge CSS in datoteke s kodo v Javascriptu.
- `script/rails` je skripta, ki se uporablja za generiranje aplikacije, modelov, krmilnikov, poganjanje spletnega strežnika WEBrick ter zagon ukazne vrstice okolja Rails. Več o njej bomo izvedeli v razdelku *Rails, Zagon aplikacije in spletnega strežnika*.
- `test/` Imenik vsebuje testne datoteke za testiranje aplikacije.
- `tmp/` Imenik vsebuje začasne datoteke.
- `vendor/` Imenik je namenjen zunanjim paketom Rails, ki niso del aplikacije same, a jih aplikacija uporablja. Če želimo na primer v svoji aplikaciji uporabljati metode iz že napisanega paketa za urejanje teksta, paket lahko namestimo na sistem, tako da bo dostopen le v naši aplikaciji. V tem primeru ga bomo namestili v imenik `vendor/`.
- `README` Datoteka vsebuje kratek opis okostja aplikacije.
- `Rakefile` Datoteka vsebuje definicijo opravil, ki jih zmore opraviti orodje `rake`.
- `Gemfile` Vsebuje definicije paketov Rails, ki jih vključuje in uporablja aplikacija. (glej razdelek *Rails, Generiranje nove aplikacije okolja Rails*)
- `config.ru` Konfiguracijska datatoka za `Rack` (orodje za prevajanje, organizacijo zahtev HTTP raznih spletnih strežnikov v Ruby metode).
- `.gitingnore` - Datoteka vsebuje vzorce, na podlagi katerih se sestavi seznam datotek, ki jih sistem kontrole verzij GIT ignorira.

3.4 Zagon aplikacije in spletnega strežnika

V prejšnjem razdelku smo že opisali, kako z generatorjem ustvarimo aplikacijo v okolju Rails. Generirano aplikacijo lahko takoj poženemo, čeprav nismo vnesli še nobene vrstice kode.

Poglejmo, kaj se zgodi.
Kot smo razložili v razdelku *Rails, Lastnosti*, se aplikacija izvaja na spletnem strežniku. Vsaka aplikacija potrebuje svoj spletni strežnik, zato se pri zagonu aplikacije zažene tudi spletni strežnik aplikacije.
V okolju Rails se pri razvijanju aplikacij uporablja spletni strežnik WEBrick. WEBrick je spletni strežnik, napisan v jeziku Ruby. Na sistem se namesti ob namestitvi programa Ruby. Pri razvijanju aplikacije strežnika WEBrick ni potrebno ustavljati in znova poganjati za vsako spremembo, ki jo naredimo v kodi. Strežnik bo spremembe avtomatsko zaznal.
Med generiranjem aplikacije smo v imeniško strukturo naše aplikacije namestili skripte, ki omogočajo zagon spletnega strežnika. Poleg zagona strežnika bodo te skripte v pravilnem zaporedju naložile in inicializirale tiste pakete okolja Rails, ki jih aplikacija potrebuje za delovanje.

Po zagonu si bomo že lahko ogledali prvo spletno stran naše aplikacije:



Slika 3. Prva stran naše aplikacije

Stran prikaže datoteko `/home/andrej/ruby/diploma/public/index.html`. Ta stran na začetku vsebuje nekaj splošnih informacij o okolju Rails.

V razdelku bomo opisali proces zagona aplikacije in spletnega strežnika. Najprej si bomo ogledali ukaz, s katerim izvedemo zagon. Nato pa bomo analizirali kodo, ki se ob zagonu izvede. Razdelek bomo razdelili na naslednja podrazdelka:

- *Kako zaženemo aplikacijo okolja Rails?*
  Tu si bomo ogledali ukaz, ki ga izvedemo, da bi pognali aplikacijo. Opisali bomo tudi posledice ukaza, to je zagon spletnega strežnika naše aplikacije. Na koncu bomo navedli še seznam spletnih strežnikov, ki so podprti v okolju Rails.
- *Analiza kode, ki se izvede pri ukazu rails server*
  V razdelku je navedena analiza kode zagona aplikacije v okolju Rails. Predstavljenih je nekaj stavčnih konstruktov jezika Ruby, ki so podrobneje razloženi v poglavju Ruby. Spoznali bomo, da je zadeva precej kompleksna in zapletena. Vendar je s stališča pisca aplikacij v okolju Rails potrebno povedati, da nam pravzaprav ni potrebno narediti praktično nič. Poženemo le ukaz `rails server`. Vse ostalo se avtomatsko zgodi v ozadju. Ker pa pogosto želimo opraviti nastavitve nekoliko drugače, je pomembno, da vseeno vemo, kaj se ob izvedbi omenjenih ukazov zgodi. Na ta način lažje prilagodimo obnašanje našega sistema, kot tudi ustrezno ukrepamo, če pride do določenih težav. Razdelek se deli na tri logične sklope, ki si sledijo v procesu zagona aplikacije in spletnega strežnika:
  - *Izvajanje metode exec_script_rails! pri ukazu rails server*
    Kot smo povedali v razdelku *Rails, Skupna koda ob izvajanju ukaza*

rails se vsi ukazi rails do ukaza `exec_script_rails!` vedejo enako. Kodo od zagona do izvršitve `exec_script_rails!` smo opisali že v tistem razdelku. Do razlike pride ob izvedbi metode `exec_script_rails!`. Ta bo sedaj pognala skripto `rails` naše aplikacije. Ta izvrši skripto `config/boot.rb`. To je prva skripta, ki uporablja le ukaze, specifične za zagon aplikacije. Lahko rečemo, da se v njej zagon naše aplikacije šele zares začne. Ukaze, ki jih izvede ta skripta `boot.rb` si bomo ogledali v naslednjem razdelku.

- *Nalaganje osnovnih paketov Rails in priprava na inicializacijo aplikacije*

  Preden se naša aplikacija dokončno zažene, je potrebno naložiti še določene pakete. Ti paketi vsebujejo metode, ki se izvršijo v zadnji stopnji zagona aplikacije. Nalaganje teh paketov se začne v skripti `boot.rb`, ko skripta naloži paket `Bundler`. Opisali bomo vlogo tega paketa. Našteli bomo najpomembnejše pakete okolja Rails, ki jih `Bundler` najde in omogoči njihovo nalaganje. Po paketu `Bundler` si bomo ogledali pripravo na naslednjo stopnjo zagona. V pripravi se naložijo osnovni paketi `ActionDispatch`, `ActiveSupport` in `Rack`. Metode teh paketov se izvajajo med zadnjo stopnjo zagona aplikacije. Opisali jo bomo v naslednjem razdelku.

- *Inicializacija aplikacije in zagon spletnega strežnika*

  Sedaj je vse pripravljeno za to, da ustvarimo osnovni objekt aplikacije in sprožimo njen zagon. Na tej stopnji zagona se ustvari objekt razreda `Rails::Server`. Objekt predstavlja spletni strežnik naše aplikacije. Z inicialiazicijo tega objekta se naložijo vsi glavni paketi okolja Rails in ustvari osnovni objekt razreda aplikacije `Diploma::Application`. Opisali bomo, kako se to zgodi, in pri tem pokazali zanimiv primer uporabe metode `inherited` v jeziku Ruby.

  Razdelek bomo zaključili z opisom metode, v kateri zaženemo tako spletni strežnik kot tudi aplikacijo. Ta metoda se imenuje `start`. Izvedemo jo na objektu razreda `Rails::Server`.

3.4.1 Kako zaženemo aplikacijo okolja Rails?

Po generiranju nove aplikacije diploma (glej razdelek *Rails, Generiranje nove aplikacije okolja Rails*) s korenskega imenika aplikacije poženemo ukaz:

```
/home/andrej/ruby/diploma$ rails server
```

V oknu terminala dobimo:

```
andrej@andrej-desktop:~/ruby/diploma$ rails server
=> Booting WEBrick
=> Rails 3.0.6 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2011-07-04 00:56:55] INFO  WEBrick 1.3.1
[2011-07-04 00:56:55] INFO  ruby 1.8.7 (2010-01-10) [i486-linux]
[2011-07-04 00:57:01] INFO  WEBrick::HTTPServer#start: pid=2449 port=3000
```

Slika 4. Zagon aplikacije

Opazimo lahko vrstico, ki vsebuje spletni naslov `http://0.0.0.0/3000`. To je privzeta spletna

stran naše aplikacije. Naslov internetnega protokola (IP) `0.0.0.0` v spletnem naslovu predstavlja IP našega računalnika (naslov bi lahko bil tudi `127.0.0.1` ali `localhost`). Vsi trije sodijo med rezervirane naslove internetnega protokola in predstavljajo naš lokalni računalnik. Številka 3000 predstavlja sistemska vrata na katerih WEBrick, spletni strežnik naše aplikacije, čaka na zahteve HTTP za našo aplikacijo.
Aplikacijo okolja Rails je mogoče konfigurirati tako, da jo izvajajo številni spletni strežniki. Pri naši aplikaciji bomo ostali pri izvajanju na privzetem spletnem strežniku WEBrick. Možne pa so tudi konfiguracije z naslednjimi spletnimi strežniki:

- Mongrel
- Apache
- lightTPD
- Capistrano

3.4.2 Analiza kode, ki se izvede pri ukazu rails server

V razdelku *Rails, Skupna koda ob izvajanju ukaza rails* smo videli, kaj se zgodi ob poljubnem zagonu ukaza `rails`. Vemo, da je ključna točka klic metode `exec_script_rails!`, ki se nahaja v razredu `Rails::ScriptRailsLoader`. V našem primeru so bile datoteke nove aplikacije že ustvarjene, med njimi pa je tudi `script/rails`. Kako se obnaša metoda `exec_script_rails!` v tem primeru, si bomo ogledali podrobneje v naslednjem razdelku.

3.4.2.1 Izvajanje metode exec_script_rails! pri ukazu rails server

Kot vemo, pri vsakem ukazu `rails` pride do izvedbe skripte `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/cli.rb`. Dogajanje do vrstice `Rails::ScriptRailsLoader.exec_script_rails!` smo opisali v razdelku *Rails, Skupna koda ob izvajanju ukaza rails*.
Oglejmo si to skripto ponovno:

```
def self.exec_script_rails!

  ...
   return unless in_rails_application? ||
                 in_rails_application_subdirectory?
   exec RUBY, SCRIPT_RAILS, *ARGV if in_rails_application?
  ...
end
```

Tokrat je pogoj `in_rails_application? || in_rails_application_subdirectory?` izpolnjen, saj smo ukaz `rails server` pognali v imeniku naše aplikacije `/home/andrej/ruby/diploma/`. In namesto, da bi se izvajanje skripte zaključilo z ukazom `return` , se sedaj izvede:

```
exec RUBY, SCRIPT_RAILS, *ARGV if in_rails_application? .
```

V našem primeru se ukaz "prepiše" v:

```
exec /usr/bin/ruby script/rails
```

Torej s programom Ruby poženenemo skripto `rails.rb`, ki je v podimeniku `script/` trenutnega imenika. Preostala koda v skripti pokriva primere, ko skripte ni mogoče najti. Datoteko `script/rails` išče navzgor po trenutni imeniški strukturi. Zato je vseeno, v katerem podimeniku krovnega imenika naše datoteke zaženemo ukaz `rails server`.

Zaradi ukaza `exec` (ki je del modula `Kernel`, glej *Ruby, Lastnosti*) se izvrševanje skripte `cli.rb` po ukazu `exec_script_rails!` konča, proces skripte `/usr/bin/ruby script/rails` pa nadaljuje na mestu našega procesa, ki smo ga pognali z `rails server`. Izvajanje programa se torej nadaljuje v skripti `/home/andrej/ruby/diploma/script/rails`. Ustvarili smo jo med generiranjem okostja aplikacije.
Oglejmo si jo:

```
#!/usr/bin/env ruby1.8

APP_PATH = File.expand_path('../../config/application',  __FILE__)

require File.expand_path('../../config/boot',  __FILE__)

require 'rails/commands'
```

Metoda `File.expand_path` s pomočjo makroja `__FILE__` sestavi absolutno pot `APP_PATH` . Ta makro vrne absolutno pot trenutne datoteke. V našem primeru bo ta pot enaka `/home/andrej/ruby/diploma/script/rails`. Metoda `expand_path` torej sestavi niz `/home/andrej/ruby/diploma/config/application` in ga shrani v spremenljivko `APP_PATH`. Spremenljivka `APP_PATH` torej predstavlja aplikacijsko skripto `/home/andrej/ruby/diploma/config/application.rb`. Uporabili jo bomo v razdelku *Inicializacija aplikacije in zagon spletnega strežnika*.
Podobno se v vrstici

```
require File.expand_path('../../config/boot',  __FILE__)
```

sestavi niz

```
/home/andrej/ruby/diploma/config/boot.
```

Niz predstavlja datoteko `/home/andrej/ruby/diploma/config/boot.rb`. Ker je pred nizom v vrstici beseda `require`, se bo na tem mestu ta skripta izvršila.
V zadnji vrstici skripte `/home/andrej/ruby/diploma/script/rails`

```
require 'rails/commands'
```

se bo izvršila skripta `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/commands.rb`.
V njej se začneta inicializacija in zagon aplikacije. Skripto bomo opisali v drugem delu naslednjega razdelka in opisovanje nadaljevali v razdelku *Inicializacija aplikacije in zagon spletnega strežnika*. Z izvajanjem bo namreč začela takoj po tem, ko bo paket `Bundler` opravil svoje delo v sklopu predpriprav na zagon. V naslednji stopnji zagona pa bo nato izvedla vse ključne ukaze za inicializacijo in zagon aplikacije.

3.4.2.2 Nalaganje osnovnih paketov Rails in priprava na inicializacijo aplikacije

Na tej stopnji zagona aplikacije moramo naložiti nekaj paketov okolja Rails. Objekte razredov teh paketov in njihove metode bomo potrebovali pri inicializaciji aplikacije in zagonu spletnega strežnika.
`Bundler` je orodje za prenos, namestitev in urejanje odvisnosti med paketi. Če pri namestitvi paketov z uporabo orodja `Bundler` navedemo paket, ki je odvisen od nekega osnovnega paketa, ki ga nismo navedli, bo `Bundler` namestil oba. `Bundler` se uporablja tudi pri izvrševanju skript jezika Ruby. (glej *Literatura in viri*.)
V našem primeru bo `Bundler` uredil odvisnosti med paketi okolja Rails, navedenimi v konfiguracijski datoteki `Gemfile`. Datoteka se nahaja v glavnem imeniku aplikacije (`home/andrej/ruby/diploma/`). V našem primeru sta v datoteki navedena le osnovni paket `rails` in paket `sqlite3`. Orodje `Bundler` bo torej poskrbelo, da se bodo naložili vsi paketi, od katerih je odvisno delovanje paketov `rails` in `sqlite3`.
Metoda `setup` paketa `Bundler` najprej prebere datoteko `Gemfile`. Ugotovi, da sta paketa `rails` in `sqlite3` odvisna od naslednjih paketov:

```
actionmailer-3.0.6.gem
actionpack-3.0.6.gem
activemodel-3.0.6.gem
activerecord-3.0.6.gem
activeresource-3.0.6.gem
activesupport-3.0.6.gem
rack-1.2.1.gem
```

Naštete pakete bomo kratko opisali na koncu tega razdelka.
Metoda `setup` nato doda poti do teh paketov v okoljsko spremenljivko `$LOAD_PATH`. Za paket `ActiveModel` denimo v to spremenljivko doda pot `/usr/lib/ruby/gems/1.8/gems/activemodel-3.0.6/lib`. To je pot, kjer je paket `ActiveModel` nameščen. Enako naredi tudi z ostalimi paketi iz seznama. Na ta način omogoči kasnejše nalaganje teh paketov.

S paketom `Bundler` smo preverili odvisnosti paketov `rails` in `sqlite3` in omogočili nalaganje vseh potrebnih paketov. S tem smo zaključili s skripto `boot.rb`.
Kot smo povedali na koncu prejšnjega razdelka, sedaj pride na vrsto skripta `rails/commands.rb`. Skripta je namenjena izvajanju različnih tipov ukaza `rails`, kot so:

- *rails console - zagon ukazne vrstice okolja rails*
- *rails generate .- zagon enega izmed podprtih generatorjev okolja rails*
- *rails server. - naš primer*

Oglejmo si del kode skripte `rails/commands.rb`, ki se nanaša na naš primer*:*

```
...
case command
...
when 'server'
  require 'rails/commands/server'
  Rails::Server.new.tap { |server|
    require APP_PATH
    Dir.chdir(Rails.application.root)
    server.start
  }
...
else
...
end
```

V skripti se v `command` zapiše niz, ki sledi besedici `rails`. S pomočjo stavka `case` izvedemo ustrezne stavke. V našem primeru je v spremenljivki `command` zapisan niz `'server'`. Zato je pogoj pri `when 'server'` izpolnjen in izvršijo se ukazi v tem bloku.
V vrstici

```
require 'rails/commands/server'
```

se izvrši skripta `server.rb`.
V tej skripti je definiran razred `Rails::Server`. Podrobneje ga bomo spoznali v naslednjem

razdelku *Inicializacija aplikacije in zagon spletnega strežnika*.
S to skripto naložimo tudi paket okolja Rails, imenovan `ActionDispatch`.

Paket `ActionDispatch` je odgovoren za usmerjanje, procesiranje zahtev in odgovorov HTTP, razčlenjevanje spletnih naslovov in podobnih stvari. Ta paket je nova komponenta v okolju Rails različice 3. Skupaj s paketoma `ActionController` in `ActionView` tvori paket `ActionPack`. Slednji je celovito odgovoren za obravnavanje zahtev HTTP. Pri tem uporablja vse tri pakete, ki ga tvorijo. Akcije, ki jih je potrebno izvesti kot posledico zahtev HTTP, so implementirane v paketu `ActionController`. Paket `ActionView` poskrbi za vsebino odgovorov HTTP na zahteve HTTP. Paket `ActionDispatch` služi za povezovanje akcij v paketu `ActionController` z ustreznimi odgovori HTTP iz paketa `ActionView`.

Skupaj s paketom `ActionDispatch` se naložita tudi paketa `ActiveSupport` in `Rack`. Paket `ActiveSupport` vključuje pomožna orodja okolja Rails, `Rack` pa vsebuje razrede, ki so odgovorni za komunikacijo s spletnim strežnikom.

Vloge paketov `ActionDispatch` in `ActiveSupport` ne bomo podrobneje predstavili. Omenimo samo, da vsebujeta nekaj metod, ki sodelujejo pri inicializaciji in zagonu aplikacije. Paket `Rack` pa bo v naslednjem razdelku odigral ključno vlogo pri zagonu aplikacije.

Prišli smo do točke inicializacije objekta naše aplikacije. Priprava je zaključena. Paket `Bundler` je razrešil odvisnosti med paketi in pripravil okolje za nalaganje paketov. Nato smo naložili vse tiste pakete okolja Rails, ki bodo pri sodelovali pri inicializaciji. Sedaj se inicializicija lahko prične.

3.4.2.3 Inicializacija aplikacije in zagon spletnega strežnika

V skripti `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/commands.rb` smo pri izvrševanju ukazov prišli do inicializacije objekta razreda `Rails::Server`. V tem objektu se bo ustvaril objekt razreda naše aplikacije. Z metodo `Rails::Server.start` pa bomo pognali inicializirani objekt in tako zagnali aplikacijo in spletni strežnik.
Oglejmo si sedaj blok kode skripte `rails/commands.rb` (glej razdelek *Ruby, Bloki in procedure*), v katerem so implementirane navedene akcije:

```
Rails::Server.new.tap { |server|
    require APP_PATH
    Dir.chdir(Rails.application.root)
    server.start
  }
```

V vrstici

```
Rails::Server.new.tap { |server|
```

se inicializira objekt razreda `Rails::Server`. Razred `Rails::Server` deduje iz razreda `Rack::Server`. Ker se v metodi `initialize` razreda `Rails::Server` kliče metoda `super`, se v njej naprej izvede inicializacija objekta razreda `Rack::Server` (glej razdelek *Ruby, Razredi, metode in moduli*).

Razred `Rack::Server` predstavlja spletni strežnik. Med inicializacijo objekta razreda `Rack::Server` se shranijo nastavitve spletnega strežnika. Na te nastavitve lahko vplivamo z dodatnimi parametri ukazu `rails server`, kot so:

```
-o <IP> - IP spletnega strežnika, privzeta vrednost je 0.0.0.0
-p <številka sistemskih vrat > - številka sistemskih vrat, na katerih
```

```
posluša strežnik, privzeta vrednost je 3000
-d - strežnik naj se zažene kot sistemski servis (daemon)
```

V prvi vrstici bloka

```
require APP_PATH
```

se izvrši skripta z imenom, katere naslov je shranjen v spremenljivki `APP_PATH`, ki jo že poznamo (glej *Izvajanje metode exec_script_rails! pri ukazu rails server*). V našem primeru gre za skripto `/home/andrej/ruby/diploma/config/application.rb`.
V tej skripti se najprej naložijo vsi paketi okolja Rails.
Med te pakete sodijo `ActionController`, `ActiveView`, `ActiveModel`, `ActionMailer` in `ActiveResource`. Pakete `ActionController`, `ActiveView`, `ActiveModel` bomo podrobneje spoznali v naslednjih poglavjih. Paketov `ActionMailer` in `ActiveResource` ne bomo uporabili v naši aplikaciji, zato le omenimo njun namen. `ActionMailer` je orodje za pošiljanje in prejemanje elektronske pošte v spletni aplikaciji. `ActiveResource` pa služi za komunikacijo z spletnimi servisi na svetovnem spletu (na primer spletna servisa Amazon in Google maps).
Posebej se naložita še osnovni paket `Rails` in paket `SQLite3`. Iz prejšnjega razdelka vemo, da sta odvisna le od paketov, naštetih zgoraj, zato ni bilo potrebno nalagati nobenih drugih dodatnih paketov.
V skripti `config/application.rb` je nato definiran modul z imenom `Diploma` in v njem razred `Application`, glavni razred naše aplikacije. Razred `Diploma::Application` deduje iz razreda `Rails::Application`. Razred `Rails::Application` pa definira metodo `inherited` (glej razdelek *Ruby, Razredi, metode in moduli*). Kot vemo, kadar kak razred definira metodo `inherited`, se ta metoda izvede vsakič, kadar je ta razred dedovan. (torej, ko se v pomnilnik naloži definicija takega razreda, ki ta razred deduje). V metodi `inherited` piše:

```
def inherited(base)

      raise "You cannot have more than one Rails::Application"
        if Rails.application

      super

      Rails.application = base.instance

 end
```

Torej se pri nalaganju definicije razreda `Diploma::Application` izvede metoda `inherited` razreda `Rails::Application`, saj razred `Diploma::Application` deduje iz razreda `Rails::Application`.
Najprej se sproži izjema, če bi slučajno poskušali zagnati že drugi primerek aplikacije Rails, saj znotraj strežnika lahko teče le ena.
V zadnji vrstici metode

```
Rails.application = base.instance
```

se izvede metoda `instance` razreda `Diploma::Application`, v kateri se ustvari nov objekt razreda `Diploma::Application`. Objekt se shrani v spremenljivko `Rails.application`. S tem smo dejansko ustvarili nov objekt naše aplikacije.

Sedaj nam preostaneta še zagon aplikacije in spletnega strežnika.
Vrstica

```
server.start
```

v skripti `rails/commands.rb` izvede zagon spletnega strežnika naše aplikacije. V metodi `start` se izvede metoda `super` (glej razdelek *Ruby, Razredi, metode in moduli*). Razred `Rails::Server`, kot vemo, deduje iz razreda `Rack::Server`. Torej se bo najprej izvedla

metoda `Rack::Server.start`. Ključna vrstica v tej metodi je:

```
server.run wrapped_app, options
```

Vrstica je videti kot klic metode `run` objekta server s parametroma `wrapped_app` in `options`. V resnici pa sta `wrapped_app` in `options` metodi tipa, ki se v Rubyju imenuje `attr_accessor` (glej razdelek *Ruby, Razredi, metode in moduli*). To pomeni, da nastavita in vrneta vrednost spremenljivk objekta razreda `Rack::Server` z imeni, enakimi imenom metod: `wrapped_app` in `options`.
Metoda `options` je relativno enostavna. Vrne nam nastavitve okolja spletnega strežnika, ki smo omenili na začetku tega razdelka (IP, številko sistemskih vrat) in jih shrani v spremenljivko `options`.
V metodi `wrapped_app` se izvede nekaj zelo pomembnih operacij. Tako se med drugim izvede metoda `parse_file`. To je metoda razreda `Rack::Builder`, ki je vključen v razredu `Rack::Server`. Metoda prebere generirano konfiguracijsko datoteko `/home/andrej/ruby/diploma/config.ru` z vsebino:

```
require ::File.expand_path('../config/environment', __FILE__)

run Diploma::Application.instance
```

Zatem se izvede inicializacija novega objekta `Rack::Builder` s parametri, kot jih določa vsebina omenjene konfiguracijske datoteke. Novi objekt `Rack::Builder` izvrši ukaze iz vsebine datoteke. Najprej izvrši skripto `/home/andrej/ruby/diploma/config/environment.rb`. Najpomembnejši ukaz v tej skripti je:

```
Diploma::Application.initialize!
```

Pri izvajanju `Diploma::Application.initialize!` se ustvarijo objekti naloženih paketov okolja Rails. Spoznali smo jih v začetku tega razdelka. Med njimi so `ActionView`, `ActionController`, `ActiveRecord`, `ActionMailer` in `ActiveResource`. Na ustvarjenih objektih se poženejo metode, imenovane inicializatorji. Ti opravijo naloge, kot so nastavitev poti v aplikaciji, vzpostavitev sistema beleženja sporočil aplikacije in nastavitev parametrov delovanja posameznih paketov. Posameznih inicializatorjev ne bomo podrobneje opisovali. Povejmo le, da smo z zagonom vseh izvršili zagon naše aplikacije.

Po končanem zagonu aplikacije metoda `parse_file` izvrši še zadnji ukaz v datoteki `config.ru`:

```
run Diploma::Application.instance
```

Metoda `run` je metoda objekta razreda `Rack::Builder`. Naslov objekta naše aplikacije shrani v tabelo, v kateri objekt razreda `Rack::Builder` hrani naslove aplikacijskih objektov. To tabelo metoda `wrapped_app` vrne kot parameter `wrapped_app` ukazu `server.run`.
Ukaz `server.run` je prav tako sestavljen iz metode tipa `attribute_accessor`, imenovane `server` in metode `run`. V metodi `server` dobimo objekt, ki predstavlja tip spletnega strežnika. V našem primeru je to objekt spletnega strežnika WEBrick. Na tem objektu se izvede metoda `run` s parametroma `wrapped_app`, ki vsebuje tabelo naslovov objektov aplikacij in `options`, ki vsebuje nastavitve spletnega strežnika. Z metodo `run` se izvrši zagon spletnega strežnika.

Razlaga zagona aplikacije in spletnega strežnika je s tem končana. Pogledali smo si vse bistvene operacije, ki se izvedejo ob zagonu. Spoznali smo tudi vlogo nekaterih ključnih paketov okolja Rails, kot je na primer paket `Rack`.

3.5 Usmerjanje

V razdelku je opisana funkcionalnost okolja Rails, imenovana usmerjanje (`Routing`). Usmerjanje je sistem za povezovanje zahtev HTTP, ki jih izvedemo v brskalniku odjemalca, z akcijami

krmilnikov naše spletne aplikacije na spletnem strežniku. Gre torej za določitev metode, ki jo bo naša aplikacija izvedla na podlagi podatkov iz spletnega naslova. Tako denimo spletni naslov `http:\\localhost:3000\paragraphs\new` sestavljata besedi `paragraphs` in `new`. `Paragraphs` predstavlja ime razreda krmilnika, `new` pa predstavlja metodo objekta razreda krmilnika `Paragraphs`. Povezava med ključnimi besedami iz spletnega naslova in metodo (akcijo) razreda krmilnika se imenuje usmeritev. Več različnih usmeritev pa sestavlja sistem usmerjanja.
Sistem usmerjanja je poleg tega odgovoren tudi za generiranje spletnih naslovov, ki predstavljajo odgovore HTTP odjemalcu na njegove zahteve HTTP naši spletni aplikaciji. Sistem povezovanja ima svojo osnovo v konfiguracijski datoteki `/config/routes.rb`. V njej uporabljena podvrsta jezika Ruby, se imenuje `Ruby Domain Specific Language` ali kratko DSL. V datoteki `/config/routes.rb` so naštete usmeritve, ki povežejo posamezno zahtevo HTTP z ustrezno akcijo krmilnika naše aplikacije. V razdelku bomo spoznali možne vnose v datoteko, ter kako se ti vnosi odražajo v spletni aplikaciji. Spoznali bomo tudi standard REST. Z upoštevanjem pravil tega standarda se pravilno definirajo usmeritve spletne aplikacije. Začeli bomo s kratko definicijo ključnih pojmov, ki jih bomo uporabljali v spletni aplikaciji, nadaljevali pa s primeri usmeritev v datoteki `config/routes.rb.` Razdelek bomo zaključili z opisom usmeritev, ki zadoščajo pravilom standarda REST. Razdelek je razdeljen na tri podrazdelke:

- *Pojmovnik osnovnih pojmov, ki jih bomo uporabljali v razdelku*
  V tem razdelku bomo navedli nekaj osnovnih definicij izrazov, ki jih bomo uporabljali v naslednjih dveh podrazdelkih.
- *Osnovni primeri usmeritev v konfiguracijski datoteki config/routes.rb*
  V tem razdelku bomo opisali različne usmeritve, ki jih lahko vnašamo v konfiguracijsko datoteko `config/routes.rb.` Na koncu bomo predstavili osnovna pravila standarda REST.
- *Primer osnovnega sistema usmeritev, ki ustreza pravilom standarda REST*
  V tem razdelku bomo predstavili skupek usmeritev, ki jih bomo uporabili pri prikazovanju spletnih strani aplikacije `diploma`. Pokazali bomo, da ustrezajo pravilom standarda REST.

3.5.1 Pojmovnik osnovnih pojmov, ki jih bomo uporabljali v razdelku

V tem razdelku bomo opisali osnovne pojme, ki jih moramo poznati, da bi lahko razumeli naslednja podrazdelka razdelka Usmerjanje.

- HTTP (Hyper Text Transfer Protocol) je mrežni protokol, na katerem je osnovan svetovni splet. Izvaja se na podlagi zahtev, ki jih odjemalec (brskalnik) poda spletnemu strežniku. Strežnik pošlje odgovor, ki se prikaže v oknu brskalnika.
- zahteva HTTP je zahteva, ki jo odjemalec preko mreže pošlje strežniku. Poznamo naslednje tipe zahtev HTTP:
  - HEAD – odjemalec zahteva glavo (header) spletne strani, brez telesa spletne strani.
  - GET – odjemalec zahteva celotno spletno stran.
  - POST – odjemalec zahteva, da strežnik sprejme vsebino zahteve HTTP
  - PUT – odjemalec zahteva, naj se vsebina zahteve HTTP shrani pod v zahtevi navedenim spletnim naslovom.
  - DELETE – odjemalec zahteva, da se spletna stran, določena s spletnim naslovom, zbriše.
  - OPTIONS – odjemalec zahteva možne tipe zahtev in njihovih nastavitev za določeni spletni naslov.
- odgovor HTTP je odgovor strežnika na zahtevo odjemalca, to je željena spletna stran v formatu HTML.
- Pot je spletni naslov posamezne spletne strani naše aplikacije (`URL – Uniform Resource`

`Locator`) in je podana kot parameter k zahtevi HTTP.

- Relativna pot je zadnji del spletnega naslova posamezne spletne strani naše aplikacije. Relativno pot sestavlja tisti del spletnega naslova, ki je dodan poti prve spletne strani naše aplikacije, kot je to npr. `http:/localhost:3000/`. Kar pride v naslovu spletne strani za tem, sestavlja relativno pot.
- Usmeritev (`route`) je povezava ene ali več poti, ki ustrezajo vzorcu opisa poti, na akcijo določenega krmilnika okolja Rails.

3.5.2 Osnovni primeri usmeritev v konfiguracijski datoteki config/routes.rb

Denimo, da v našo spletno aplikacijo pride zahteva HTTP:

```
GET http://localhost:3000/paragraphs/
```

Uporabnik je torej v brskalniku zahteval vsebino spletne strani `http://localhost:3000/paragraphs/`. Iz spletnega naslova vidimo, da jo bo prestregel naš strežnik WEBrick, ki posluša pri sistemskih vratih 3000. Tu se vključi sistem usmeritev, ki v datoteki `config/routes.rb` poišče usmeritev, ki ustreza tej poti.
Denimo, da najde usmeritev:

```
match 'paragraphs', :to => 'paragraphs#index'
```

Metoda `match` je definirana z dvema parametroma. Prvi je vedno relativna pot spletnega naslova, drugi pa je slovar opcij.
Opcija `:to` predstavlja tarčo, s katero je povezana relativna pot, določena v prvem parametru. Tarča je ponavadi določena v notaciji `krmilnik#akcija`.
Zgornja zahteva HTTP bo tako sprožila akcijo `ParagraphsController::index`. Ta akcija pa bo kot odgovor HTTP vrnila spletno stran `http://localhost:3000/paragraphs/index.html`.
Usmeritev z enakim rezultatom bi lahko zapisali tudi v skrajšani obliki z uporabo slovarja:

```
match 'paragraphs' => 'paragraphs#index'
```

Osnovni primer, kot je naš, bi celo lahko zapisali kot:

```
match 'paragraphs/index'
```

Vendar slednja notacija velja le v primeru, ko je relativna pot sestavljena le iz dveh segmentov. Prvi segment predstavlja krmilnik, drugi pa akcijo.
Rezultat dodane usmeritve lahko vedno preverimo z ukazom iz ukazne vrstice:

```
rake routes
```

Za obširno razlago uporabe orodja `Rake` glej razdelek ***Rails, Rake***.
Ukaz nam vrne vse usmeritve, ki so definirane v naši aplikaciji. V našem primeru po izvedbi ukaza `rake routes` dobimo:

```
paragraphs        /paragraphs(.:format)
{:controller=>"paragraphs", :action=>"index"}
```

Na prvem mestu je ime usmeritve `paragraphs`, ki se je v našem primeru dodalo avtomatično. Ime usmeritve sicer določimo z uporabo opcije `:as (:as =>'paragraphs')`. Z uporabo opcije `as` smo definirali tudi dve uporabni metodi krmilnika:
Metoda `paragraphs_path` vrne relativno pot spletnega naslova, v našem primeru `paragraphs/index`. Metoda `paragraphs_url` vrne pot spletnega naslova, v našem primeru `http://localhost:3000/paragraphs/index`.
Naslednja opcija, ki jo lahko dodamo usmeritvi, je `:via`:

```
match 'paragraphs' => 'paragraphs#index', :via => :get
```

Ukaz `rake routes` nam vrne:

```
paragraphs GET    /paragraphs(.:format)
{:controller=>"paragraphs", :action=>"index"}
```

Z opcijo `:via` smo določili za kakšen tip zahteve HTTP velja usmeritev, v našem primeru je to le za zahteve HTTP tipa GET. Lahko pa bi, denimo, dodali opcijo `:via => [:get,:post]`. V tem primeru bi usmeritev veljala tako za zahteve HTTP tipa GET kot tudi POST.
Na zadnjem primeru smo opazili, da lahko ena usmeritev velja za več različnih zahtev HTTP. Če pa opcije `:via` ne dodamo, potem velja usmeritev kar za vse tipe zahtev.
Ker shranjujemo podatke naše aplikacije v bazo podatkov in ima vsak podatek svojo identifikacijsko številko (`id`), se mora pri prikazovanju posameznega podatka iz baze `id` pojaviti tudi v poti spletnega naslova.

```
http://localhost:3000/paragraphs/1
```

je denimo prvi razdelek v tabeli razdelkov. Številka 1 predstavlja `id` razdelka.
Če bi za vsak razdelek vnesli svojo usmeritev v datoteko `config/routes.rb`, npr kot:

```
match 'paragraphs/1' => 'paragraphs#1', :via => :get

match 'paragraphs/2' => 'paragraphs#2', :via => :get

match 'paragraphs/3' => 'paragraphs#3', :via => :get
```

bi datoteka hitro postala nepregledna. Za rešitev problema bomo uporabili spremenljive parametre usmeritve. V našem primeru bi tako problem rešili z usmeritvijo:

```
match 'paragraphs/:id(.:format)' => 'paragraphs#show'
```

`:id` in `:format` sta primera spremenljivih parametrov usmeritve. Ob razbiranju poti spletnega naslova `http://localhost:3000/paragraphs/1.html` se v simbol `:id` nastavi številka `1` in simbol `:format` bo postal `html`. Parametra bosta dostopna tudi v akciji krmilnika `ParagraphsController::show` v tabeli parametrov kot `params[:id]` in `params[:format]`.
Usmeritve, ki smo jih podali v zgornjih primerih, so prikazale eno od glavnih pravil standarda REST. (`Represetational State Transfer`). Vsak del poti spletnega naslova predstavlja nek podatek. `Localhost` je naslov našega računalnika, `3000` so sistemska vrata na katerih "posluša" strežnik WEBrick, `paragraphs` je tabela razdelkov, `1` je prvi vnos v tabeli razdelkov. Nič od naštetega ne predstavlja nekega abstraktnega tipa ali metode.
Pravila standarda REST so:

- Vsak pomemben del poti predstavlja objekt.
- Vsak objekt je poimenovan.
- Na objektih se lahko izvaja standardne operacije po načelu CRUD:
  - Create – ustvari objekt
  - Read – preberi objekt
  - Update – spremeni objekt
  - Delete – zbriši objekt
- Odjemalec in strežnik komunicirata z določeno množico tipov zahtev in odgovorov.

Odjemalec zahteva, strežnik odgovori in komunikacija se konča.

Ogledali smo si osnovne usmeritve v datoteki `config/routes.rb` ter pravila standarda REST. V okolju Rails je bistvo hitrega razvoja aplikacij dejstvo, da lahko z eno dodano vrstico kode takoj dobimo kompletno funkcionalnost določenega dela aplikacije. To velja tudi za usmerjanje. Tak primer si bomo ogledali v naslednjem razdelku.

3.5.3 Primer osnovnega sistema usmeritev, ki ustreza pravilom standarda REST

Denimo, da vnesemo v datoteko `config/routes.rb` vrstico:

```
resources :paragraphs
```


Z ukazom `rake routes` lahko preverimo, kakšne usmeritve smo dobili za krmilnik

```
paragraphs
        paragraphs GET    /paragraphs(.:format)
      {:controller=>"paragraphs", :action=>"index"}

                          POST   /paragraphs(.:format)
      {:controller=>"paragraphs", :action=>"create"}

       new_paragraph GET    /paragraphs/new(.:format)
      {:controller=>"paragraphs", :action=>"new"}

      edit_paragraph GET    /paragraphs/:id/edit(.:format)
      {:controller=>"paragraphs", :action=>"edit"}

       paragraph GET    /paragraphs/:id(.:format)
      {:controller=>"paragraphs", :action=>"show"}

                        PUT    /paragraphs/:id(.:format)
      {:controller=>"paragraphs", :action=>"update"}

                        DELETE /paragraphs/:id(.:format)
      {:controller=>"paragraphs", :action=>"destroy"}
```

Z eno vrstico smo torej nastavili praktično vse usmeritve, ki jih standardna aplikacija potrebuje za delo s podatki v tabeli razdelkov. Prvo usmeritev smo predstavili v prejšnjem razdelku. Namenjena je prikazovanju kazala.
Usmeritev `POST  /paragraphs(.:format) {:controller=>"paragraphs", :action=>"create"}` je primer usmeritve zahteve HTTP tipa POST. V standardni aplikaciji se ta zahteva izvede takrat, ko imamo na spletni strani nek obrazec (formo) in želimo vnešene podatke shraniti s klikom na gumb "`Submit`". Ker se podatki vnašajo in ne berejo, je zahteva drugačnega tipa, torej POST in ne GET.
Usmeritev `new_paragraph GET  /paragraphs/new(.:format) {:controller=>"paragraphs", :action=>"new"}` nas usmeri na spletno stran z obrazcem za vnos novega podatka v bazo.
Usmeritev `edit_paragraph GET  /paragraphs/:id/edit(.:format) {:controller=>"paragraphs", :action=>"edit"}` nas usmeri na spletno stran z obrazcem za spreminjanje podatka v bazi.
Usmeritev `paragraph GET  /paragraphs/:id(.:format) {:controller=>"paragraphs", :action=>"show"}` smo spoznali v prejšnjem razdelku. Namenjena je prikazovanju podatka z identifikacijsko številko `:id` iz baze podatkov.
Usmeritev `PUT  /paragraphs/:id(.:format) {:controller=>"paragraphs", :action=>"update"}` je namenjena spreminjanju podatka z identifikacijsko številko `:id` v bazi. Ta usmeritev se izvede, ko na neki spletni strani, ki vsebuje obrazec, kliknemo na gumb "`Submit`".
Zadnja usmeritev `DELETE /paragraphs/:id(.:format) {:controller=>"paragraphs", :action=>"destroy"}` je namenjena izbrisu podatka z identifikacijsko številko `:id` iz baze podatkov.

Pri pozornem opazovanju se jasno vidi, da so izpolnjena načela CRUD (Ustvari, Preberi, Posodobi, Zbriši) standarda REST.
V datoteki `config/routes.rb` je pomemben tudi vrstni red usmeritev. Prva se obravnava usmeritev na začetku datoteke, nato pa ji zaporedoma sledijo tiste pod njo. Ko se najde usmeritev, ki ustreza poti spletnega naslova in tipu zahteve HTTP, se izvede akcija krmilnika določenega v usmeritvi. Če obstaja pod njo še ena drugačna usmeritev, ki tudi ustreza poti in tipu zahteve HTTP, se ta ne bo izvedla, čeprav bi se morda morala.

S tem smo zaključili z razdelkom Usmerjanje. Opisali smo, kaj usmeritve so in čemu služijo. Nato smo predstavili več načinov dodajanja usmeritev in navedli pravila stanadarda REST. V zadnjem razdelku smo videli, kako z eno samo dodano vrstico `:resources paragraphs`

dodamo vse potrebne usmeritve za prikazovanje spletnih strani tipičnih akcij krmilnika. Te akcije bomo podrobneje spoznali v razdelku *Rails, Opis generiranja krmilnika in njegovih tipičnih akcij*.

3.6 Konfiguracija baze podatkov

Razdelek opisuje konfiguracijo baze podatkov. Ta se izvede po generiranju nove aplikacije v okolju Rails. Povedali bomo, kako pri generaciji aplikacije specifiramo tip baz podatkov. Ogledali si bomo vsebine nekaterih ključnih konfiguracijskih datotek baz podatkov aplikacije. Razdelek bomo zaključili z opisom ukaza, s katerim kreiramo novo bazo podatkov. Ta postopek bomo podrobneje spoznali v naslednjem razdelku.

V našem primeru pri generiranju aplikacije nismo izbrali sistema za upravljanje baze podatkov. Glede na privzete nastavitve smo s tem dobili RDBMS SQLite. Ker želimo uporabljati sistem MySql, bomo to nastavitev sedaj spremenili. Če bi želeli to postoriti že pri generiranju aplikacije, bi morali pognati ukaz v obliki `rails new diploma -d mysql` ali v daljši obliki `rails new diploma --database=mysql`.
Generiranje aplikacije smo spoznali v razdelku *Rails, Generiranje nove aplikacije okolja Rails*. V razdelku smo omenili konfiguracijsko datoteko za baze podatkov `config/database.yml`. Kot smo povedali v omenjenem razdelku, ta datoteka vsebuje podatke za konfiguracijo baz podatkov naše aplikacije.
Oglejmo si del njene vsebine:

```
# SQLite version 3.x
#   gem install sqlite3
development:
  adapter: sqlite3
  database: db/development.sqlite3
...
```

Obstajajo trije načini delovanja, v katerih lahko deluje naša aplikacija. To so `development`, `test` in `production`. `Development` uporabljamo pri razvoju aplikacije. `Test` uporabljamo pri testiranju. V načinu `production` pa delujejo končane aplikacije. Aplikacija lahko v vsakem načinu uporablja drugo bazo podatkov. V načinu `development` uporablja bazo z imenom `diploma_development`. V načinu `test` uporablja bazo `diploma_test` in v načinu `production` uporablja `diploma_production`. Nastavitev načina vidimo v tretji vrstici zgornjega dela vsebine datoteke `database.yml`. Pri razvoju naše aplikacije bomo uporabljali način `development` in bazo podatkov `diploma_development`.
V naslednji vrstici konfiguracijske datoteke

```
adapter: sqlite3
```

je določen tip baze podatkov, oziroma sistem za upravljanje z bazami podatkov. Ker smo pri generiranju uporabili privzete nastavitve, je tip baze `sqlite3`. V vrstici

```
database: db/development.sqlite3
```

je specifirana datoteka, ki vsebuje bazo tipa `sqlite3` na sistemu. Baze tipa `sqlite3` hranijo v podatke kar v datoteki s končnico `.sqlite3`.
V imeniku `/home/andrej/ruby/` želimo spremeniti konfiguracijo baze podatkov za našo aplikacijo. Še enkrat bomo generirali našo aplikacijo, tokrat z opcijo `--database=mysql`. To lahko storimo, ker je naša aplikacija trenutno le okostje in baz podatkov še nismo uporabili.
Poženemo ukaz:

```
rails new diploma -d mysql
```


Dobimo sporočilo:

```
conflict  config/database.yml
Overwrite /home/andrej/ruby/diploma/config/database.yml? (enter "h" for
help) [Ynaqdh] Y
```

Pri generiranju je skripta opazila, da datoteka `config/database.yml` že obstaja in da je njena vsebina drugačna od vsebine novo generirane skripte z istim imenom. Zato nas sprašuje, kaj naj stori.
Zadevo razrešimo z odločitvijo za prepis datoteke in vtipkamo `Y`. Dobimo novo vsebino datoteke `database.yml`. Oglejmo si jo:

```
# Install the MySQL driver:
#   gem install mysql2
...
development:
  adapter: mysql2
  ...
  database: diploma_development
  ...
  username: root
  password
```

Tip baze se je zamenjal v `mysql`. Baze podatkov tipa `mysql` hranijo podatke drugače kot baze tipa `sqlite3`, zato je v nastavitvi `database` tokrat ime baze in ne pot do datoteke s podatki. Opazimo tudi novi nastavitvi za uporabnika in geslo, ki ju bomo morali pravilno nastaviti.
Pri drugem generiranju smo prepisali še eno konfiguracijsko datoteko `Gemfile`. Spoznali smo jo v razdelku *Rails, Generiranje nove aplikacije okolja Rails*. Uporabljalo jo je orodje `Bundler`. V njej sta bila navedena le dva paketa `rails` in `SQlite3`. Če si jo ogledamo sedaj, ugotovimo, da je paket `SQLite3` zamenjal paket `MySQL2` (številka 2 na koncu paketa predstavlja različico tega paketa). Če želimo torej uporabljati drug RDBMS moramo namestiti primeren paket Ruby, ki zna sodelovati s tem sistemom. V našem primeru je to sedaj Ruby paket `MySQL2`. Ta paket ponavadi namestimo kar ob namestitvi okolja Rails, če že ni nameščen.

Nastavitev uporabnika in gesla v datoteki `database.yml` je ob nameščenem paketu `MySQL2` pravzaprav vse, kar moramo narediti za začetek dela z bazami podatkov aplikacije. Kako nastavimo geslo za uporabnika `root`, si preberemo v dokumentaciji nastavitev jezika za delo z bazami podatkov `mysql` (glej *Literatura in viri*). Lahko pa uporabljamo uporabnika `root` tudi brez gesla (privzeta nastavitev).

Razdelek bomo zaključili z kreiranjem baze podatkov `diploma_development`, kar bo iztočnica za naslednji razdelek. Bazo podatkov kreiramo z orodjem `rake` tako, da v imeniku `/home/andrej/ruby/diploma/` poženemo ukaz:

```
rake db:create
```

Da bi preverili rezultat ukaza, bomo uporabili izvršilno vrstico jezika za delo z bazami podatkov `mysql`. Poženemo ukaz `mysql`

```
~/ruby/diploma# mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 38
...
mysql>
```

in vtipkamo naslednje ukaze:

```
show databases
| diploma_development      |
|
+--------------------------+
mysql> use diploma_development;
Database changed
mysql> show tables;
Empty set (0.00 sec)
```

Naredili smo torej bazo podatkov z imenom `diploma_development`. Baza ne vsebuje nobenih tabel. Kako v okolju Rails kreiramo podatke, bomo ugotovili v razdelku *Razvoj aplikacije, Določitev podatkov v bazi podatkov in generiranje modelov*. V naslednjem razdelku se bomo posvetili orodju `rake`. Med drugim si bomo pogledali tudi to, kako orodje `rake` kreira bazo podatkov aplikacije.

3.7 Rake

Ukaz `rake` je Rubyjeva oblika ukaza `make`. `Make` je ukaz, ki ga v operacijskih sistemih družine Unix uporabljamo za generiranje izvršilnih datotek. Ukaz `rake` ima veliko pomembnih funkcij. Med drugim omogoča kreiranje baze podatkov in izvajanje sprememb (migracij) na bazi.

- *rake db:create kreiranje*
- *rake db:migrate migracije*
- *rake routes izpiše obstoječe povezave med zahtevami HTTP za določene strani HTML, ter akcijami krmilnika.*

V razdelku bomo najprej opisali, kako ukaz `rake` sploh deluje. Ta za svoje delovanje uporablja konfiguracijske datoteke s končnico `.rake`. V teh datotekah so definirana opravila, ki jih lahko izvedemo z orodjem `rake`. Opravili z imeni `db:create` in `routes` sta primera takih opravil, definiranih v okolju Rails. Po opisu splošne definicije opravil se bomo posvetili ukazoma `db:create` in `db:migrate`.
Na koncu razdelka si bomo pogledali še ukaz s katerim `rake` vrne usmeritve, zapisane v datoteki `config/routes.rb`. Razdelek bomo razdelili na naslednje logične sklope:

- *Osnovna uporaba orodja rake*
  V tem razdelku si bomo ogledali, kako definiramo opravila, ki jih potem izvedemo z orodjem `rake`. Videli bomo primere gnezdenih opravil. Opravila, ki jih `rake` uporablja v okolju Rails, so večinoma gnezdena. V tem razdelku bomo osvojili znanje, ki ga bomo uporabili v naslednjih dveh razdelkih pri opisovanju uporabe orodja `rake` v okolju Rails.
- *Analiza kode, ki se izvede ob ukazih rake db:create in rake db:migrate*
  V tem razdelku bomo opisali ukaze, ki se izvedejo ob kreaciji baze. Spoznali bomo konfiguracijsko datoteko okolja Rails `/usr/lib/ruby/gems/1.8/gems/activerecord-3.0.6/lib/active_record/railties/databases.rake`, ki jo orodje `rake` uporablja med kreacijo baze in izvajanjem migracij na bazi podatkov. Ogledali si bomo glavni razred za delo z bazami podatkov, razred `ActiveRecord::Base`. Tega `rake` uporabi, da ustvari bazo podatkov.
- *Analiza kode, ki se izvede ob ukazu rake routes*
  V tem razdelku bomo spoznali način izpisovanja vseh usmeritev, definiranih v datoteki `config/routes.rb`. Usmeritve smo podrobneje spoznali v razdelku *Rails,*

Usmerjanje.

3.7.1 Osnovna uporaba orodja rake

V uvodu smo povedali, da je `rake` Ruby oblika ukaza `make`. V operacijskem sistemu iz družine Unix z ukazom `make` ponavadi le gradimo izvršilne datoteke, z ukazom `rake` pa lahko opravljamo najrazličnejša opravila. Podobnost med obema ukazoma se kaže v tem, da oba za delovanje uporabljata konfiguracijske datoteke. V Unixu pri ukazu `make` ponavadi uporabljamo konfiguracijsko datoteko `Makefile`, pri ukazu `rake` pa se konfiguracijska datoteka imenuje `Rakefile`. Osnovna konfiguracijska dakoteka `Rakefile` je napisana v jeziku Ruby.
Oglejmo si preprost primer uporabe orodja `rake`. V praznem imeniku `/home/andrej/testrake/` kreiramo datoteko `Rakefile` z naslednjo vsebino:

```
task :testno_opravilo do

    puts "Testno opravilo rake"

end
```

Z besedo `task` smo definirali novo opravilo. Simbol `:testno_opravilo` (glej razdelek *Ruby, Simboli*), ki ji sledi, je ime tega opravila. Uporabljali ga bomo pri klicu ukaza `rake`. Nato sledi blok ukazov, ki jih bomo v opravilu izvedli. V našem primeru bomo samo izpisali testno sporočilo.
V imeniku `/home/andrej/testrake/` poženemo ukaz:

```
root@andrej-desktop:~/ruby/testrake# rake testno_opravilo

(in /home/andrej/ruby/testrake)

Testno opravilo rake
```

Opravilom lahko definiramo tudi parametre. V ta namen v datoteko `Rakefile` dodajmo še vrstice:

```
task :test_s_parametrom do

    parameter = ENV["PARAM"] || "parameter"

    puts "Testno opravilo s parametrom #{parameter}"

  end
```

V bloku ukazov smo definirali parameter `PARAM` s privzeto vrednostjo "`parameter`". V bloku ukazov nato le izpišemo sporočilo s prejetim parametrom. Če pokličemo opravilo `test_s_parametrom` brez parametra, se bo izpisala privzeta vrednost:

```
root@andrej-desktop:~/ruby/testrake# rake test_s_parametrom

(in /home/andrej/ruby/testrake)

Testno opravilo s parametrom parameter
```

Parameter pa lahko določimo z ukazom:

```
root@andrej-desktop:~/ruby/testrake# rake PARAM="testparam"
test_s_parametrom

(in /home/andrej/ruby/testrake)

Testno opravilo s parametrom testparam
```

Posamezna opravila lahko tudi združujemo. V obstoječo datoteko `Rakefile` dodajmo še:

```
task :sestavljeno_opravilo => [ :testno_opravilo, :test_s_parametrom ]
do

    puts "Opravili smo dve opravili"

end
```


V opravilo `sestavljeno_opravilo` smo vključili obe zgornji opravili. Ob klicu sestavljenega

opravila se bosta izvedli pred blokom ukazov sestavljenega opravila:

```
root@andrej-desktop:~/ruby/testrake# rake sestavljeno_opravilo
(in /home/andrej/ruby/testrake)
Testno opravilo rake
Testno opravilo s parametrom parameter
Opravili smo dve opravili
```

Poleg sestavljenih opravil pa lahko pri opravilih uporabimo še poimenovane prostore. Z uporabo poimenovanih prostorov opravila združujemo po namembnosti.

```
namespace :modul_opravil do
   desc "Testno opravilo"
   task :testno_opravilo do
     puts "Testno opravilo rake"
   end
   desc "Testno opravilo s parametrom"
   task :test_s_parametrom do
      parameter = ENV["PARAM"] || "parameter"
      puts "Testno opravilo s parametrom #{parameter}"
   end
   desc "Testno opravilo z objektom"
   task :test_z_razredom do
     Test.new.test
   end
end
```

Poimenovane prostore označimo z besedo `namespace`, ki ji sledi simbol z imenom poimenovanega prostora. V našem primeru se poimenovani prostor imenuje `modul_opravil`. Posamezna opravila poimenovanega prostora definiramo v bloku ukazov poimenovanega prostora.
Z ukazom:

```
root@andrej-desktop:~/ruby/testrake# rake
modul_opravil:test_s_parametrom
(in /home/andrej/ruby/testrake)
Testno opravilo s parametrom parameter
```

pokličemo opravilo `test_s_parametrom` poimenovanega prostora `modul_opravil`. V naslednjem razdelku bomo videli, da na zelo podoben način delujeta ukaza `rake db:create` in `rake db:migrate`.
Ukaz `desc` nam omogoča opisovanje opravil. Če želimo dobiti opis opravil, definiranih v datoteki `Rakefile`, izvedemo ukaz:

```
root@andrej-desktop:~/ruby/testrake# rake -T
(in /home/andrej/ruby/testrake)
rake modul_opravil:test_s_parametrom  # Testno opravilo s parametrom
rake modul_opravil:testno_opravilo    # Testno opravilo
root@andrej-desktop:~/ruby/testrake# rake modul_opravil:test_z_razredom
```


Seveda s tem dobimo le opis tistih opravil, ki smo jim z vrstico `desc "opis opravila"`

definirali opis opravila.
V datoteki `Rakefile` lahko poleg opravil definiramo tudi razred ali vključimo poljuben modul. V našem primeru smo kar v datoteki `Rakefile` definirali razred:

```
class Test

  def test

    puts "Testna metoda"

  end

end
```

V opravilu `test_z_razredom` smo ustvarili objekt razreda `Test` in poklicali njegovo metodo `test`. S tem smo dokazali, da v datoteki lahko definiramo razrede ali po potrebi tudi vključimo poljubno skripto, napisano v jeziku Ruby.

V tem razdelku smo torej spoznali, kako deluje orodje `rake`. Pogledali smo si načine, na katere definiramo in združujemo posamezna opravila. Na koncu smo videli, da v datoteki lahko uporabimo poljubne konstrukte jezika Ruby. V naslednjem razdelku si bomo ogledali, kako sta v okolju Rails implementirana ukaza `rake db:create` in `rake db:migrate`. Pri tem bomo uporabili znanje, pridobljeno v tem razdelku.

3.7.2 Analiza kode, ki se izvede ob ukazih rake db:create in rake db:migrate

V tem razdelku bomo spoznali kodo, ki se izvede ob ukazu `rake db:create`. V prejšnjem razdelku smo izvedeli, da ukaz `rake` uporablja konfiguracijsko datoteko `Rakefile`. Ta datoteka se nahaja v imeniku `/home/andrej/ruby/diploma/`, v katerem smo v razdelku *Rails, Konfiguracija baze podatkov* pognali ukaz `rake db:create`. Oglejmo si vsebino datoteke `Rakefile`:

```
require File.expand_path('../config/application', __FILE__)

require 'rake'

Diploma::Application.load_tasks
```

V prvi vrstici naložimo skripto `/home/andrej/ruby/diploma/config/application.rb`. To skripto smo spoznali v razdelku *Rails, Zagon aplikacije in spletnega strežnika*. V njej je definiran razred naše aplikacije `Diploma::Application`, ki deduje iz razreda `Rails::Application`. V vrstici `require 'rake'` naložimo krovni modul razredov orodja `rake`. V zadnji vrstici poženemo metodo razreda `Rails::Application` `load_tasks`. V tej metodi se naložijo opravila vseh standardnih paketov okolja Rails. Med njimi je tudi paket `ActiveRecord`. Za naš primer je pomemben, ker vsebuje datoteko `/usr/lib/ruby/gems/1.8/gems/activerecord-3.0.6/lib/active_record/railties/databases.rake`. V njej se nahaja opravilo poimenovanega prostora `db` imenovano `:create`. Iz prejšnjega razdelka pa vemo, da se ob zagonu ukaza `rake db:create` izvrši opravilo `:create` poimenovanega prostora `db`. Oglejmo si del vsebine datoteke `databases.rake`, ki se nanaša na naše opravilo:

```
task :create => :load_config do

   ...

   create_database(ActiveRecord::Base.configurations[Rails.env])

end
```

Vidimo, da je opravilo `:create` sestavljeno. Pred njem se bo izvedlo še opravilo `:load_config`, ki je definirano v isti datoteki:

```
task :load_config => :rails_env do
   require 'active_record'
```

```
    ActiveRecord::Base.configurations =
    Rails.application.config.database_configuration
end
```

Opravilo `:load_config` je tudi sestavljeno, najprej se bo izvršilo opravilo `:rails_env`. To opravilo je definirano v datoteki `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/tasks/misc.rake`:

```
task :rails_env do
  unless defined? RAILS_ENV
    RAILS_ENV = ENV['RAILS_ENV'] ||= 'development'
  end
end
```

Kot vidimo, opravilo definira okoljsko spremenljivko `RAILS_ENV`, ki ji nastavi vrednost na `"development"`. Vrednost predstavlja privzeti način, v katerem deluje naša aplikacija in hkrati določa ime baze naše aplikacije `diploma_development`. Ko je opravilo `:rails_env` izvršeno, se v vrstici `require 'active_record'` v opravilu `:load_config` naloži krovni modul razredov paketa `ActiveRecord`. Med njimi je tudi osnovni razred `ActiveRecord::Base`. V zadnji vrstici opravila se v spremenljivko razreda `ActiveRecord::Base.configurations` shranijo konfiguracijske opcije baz podatkov naše aplikacije. Te se, kot vemo iz razdelka *Rails, Konfiguracija baze podatkov*, nahajajo v konfiguracijski datoteki `config/database.yml`.

Ko se konča opravilo `:load_config`, se požene blok ukazov opravila `:create`. Najpomembnejša vrstica v tem bloku je `create_database(ActiveRecord::Base.configurations[Rails.env])`. V njej izvedemo metodo `create_database`. Kot parameter ji podamo konfiguracijske nastavitve baze podatkov `diploma_development`. Slednje najdemo v konfiguracijski datoteki `config/database.yml`. Oglejmo si del vsebine te metode, ki se nanaša na naš tip baze podatkov:

```
def create_database(config)
  ...
  case config['adapter']
      when /mysql/
  ...
  ActiveRecord::Base.establish_connection(config.merge(
 ' database' => nil))
  ActiveRecord::Base.connection.create_database(config[
  'database'], creation_options)
  ActiveRecord::Base.establish_connection(config)
  ...
```

Z metodo `create_database` ustvarimo bazo podatkov `diploma_development`. Parameter `config['database']` poznamo iz razdelka *Rails, Konfiguracija baze podatkov* iz opisa datoteke `config/database.yml`. V našem primeru je ta parameter enak nizu `"development"`. Drugi parameter je slovar `creation_options`, ki vsebuje nastavitvi nabora in tipa znakov, ki jih bomo v bazi podatkov uporabljali. V našem primeru bomo uporabljali znake tipa Unicode (UTF-8). Z metodo razreda `establish_connection` se povežemo z novo ustvarjeno bazo podatkov `diploma_development`.

Sedaj si bomo ogledali še ukaz, s katerim ustvarimo tabele v bazi podatkov. Z ukazom `rake db:migrate` bazo namreč spreminjamo. Ob vsaki migraciji se v bazi podatkov izvede sprememba. Ta sprememba lahko pomeni kreiranje baze podatkov, dodajanje ali odvzemanje posameznih stolpcev ali spreminjanje njihovega tipa. Ko poženemo ukaz `rake db:migrate` se izvede opravilo z vsebino:

```
task :migrate => :environment do

    ActiveRecord::Migration.verbose = ENV["VERBOSE"] ? ENV["VERBOSE"]
    == "true" : true

    ActiveRecord::Migrator.migrate("db/migrate/", ENV["VERSION"] ?
    ENV["VERSION"].to_i : nil)

    Rake::Task["db:schema:dump"].invoke if
    ActiveRecord::Base.schema_format == :ruby

end
```

Vidimo, da je tudi `:migrate` sestavljeno opravilo. Pred blokom ukazov opravila `:migrate` se bo izvedlo opravilo `:environment`. To opravilo naloži datoteko `/home/andrej/ruby/diploma/config/environments/development.rb`, če le ta obstaja. V tej datoteki lahko nastavimo konfiguracijske opcije naše aplikacije v načinu delovanja `development`. Z nastavitvijo opcij v tej datoteki lahko spremenimo konfiguracijske opcije naše aplikacije, ki so sicer definirane v datoteki `config/application.rb`. V ukazu bloka opravil se najprej nastavi beleženje sporočil. Z uporabo parametra `VERBOSE=true` pri ukazu `rake db:migrate` bi dobili več sporočil ob izvajanju programa.
V naslednjem ukazu `ActiveRecord::Migrator.migrate("db/migrate/", ENV["VERSION"] ? ENV["VERSION"].to_i : nil)` se dejansko izvedejo migracije. Te bomo obširneje razložili takoj po razlagi zadnjega ukaza. Z `Rake::Task["db:schema:dump"].invoke if ActiveRecord::Base.schema_format == :ruby` se v datoteko `/home/andrej/ruby/diploma/db/schema.rb` shrani shema naše baze podatkov po izvedbi migracij. V primeru selitve baze podatkov na drug sistem z njo spravimo novo bazo v stanje enako stanju baze na starem sistemu. Tako nam pri selitvi ni treba izvajati migracij.
Oglejmo si sedaj, kaj se zgodi ob ukazu `ActiveRecord::Migrator.migrate("db/migrate/", ENV["VERSION"] ? ENV["VERSION"].to_i : nil)`. Vidimo, da se izvede metoda razreda `ActiveRecord::Migrator` imenovana `migrate`. Kot parametra sta ji podana niz `"db/migrate"` in verzija izvršilne datoteke Ruby, v našem primeru 1.8.7. Parameter `"db/migrate"` predstavlja imenik naše aplikacije `/home/andrej/ruby/diploma/db/migrate/`. Imena vseh datotek v tem imeniku se začnejo s časovno značko, ki označuje čas njihovega nastanka. Metoda `migrate` izvede le tiste spremembe, ki so definirane v datotekah, ki so bile ustvarjene po tem, ko smo opravili zadnjo migracijo, že izvedeno na bazi podatkov. Ker v našem primeru na bazi še ni bilo izvedene nobene migracije, metoda izvede spremembe, navedene v zaenkrat edini migracijski datoteki `<časovna_značka>_create_paragraphs.rb`. To smo ustvarili, ko smo kreirali model tabele razdelkov (glej razdelek *Rails, Model*). Oglejmo si njeno vsebino:

```
class CreateParagraphs < ActiveRecord::Migration

  def self.up

    create_table :paragraphs do |t|

      t.integer :parent_id

      t.string :type_paragraph

      t.string :title

      t.text :contents
```

```
      t.string :number
      t.timestamps
    end
  end

  def self.down
    drop_table :paragraphs
  end
end
```

V njej definiran razred `CreateParagraphs`, v katerem definiramo metodi `up` in `down`. Metoda `up` se izvede ob izvedbi migracije in ustvari ustrezno tabelo, metoda `down` pa nam povrne prejšnje stanje (pred migracijo). V metodi `up` definiramo posamezne stolpce in tipe stolpcev tabele razdelkov. Vsaka tabela ima tri privzete stolpce `id`, `created_at` in `updated_at`, ki jih metoda `create_table` ustvari avtomatsko. `id` je primarni ključ tabele razdelkov, v katerem se hranijo identifikacijske številke razdelkov. Z ukazom `t.timestamps` pa ustvarimo stolpca `created_at` in `updated_at`. Prvi bo hranil časovno značko nastanka posameznega vnosa v tabelo razdelkov, drugi pa časovno značko zadnje spremembe posameznega razdelka v tabeli razdelkov.
Ko izvedemo ukaz `rake db:migrate`, se izvede metoda `create_table` in v naši bazi podatkov se ustvari tabela razdelkov.
Oglejmo si še nekaj ukazov, ki jih uporabljamo pri naslednjih migracijah. Stolpce v tabeli lahko dodajamo, odstranjujemo in spreminjamo tipe stolpcev. Kadar želimo na bazi podatkov izvesti novo migracijo, pokličemo generator migracij:

```
rails generate migration <ime_migracije>
```

V imeniku `db/migrate/` se generira nova datoteka s trenutno časovno značko in imenom migracije. Tu sta tudi avtomatsko definirani metodi razreda `up` in `down`, ki smo ju spoznali malo prej. Vanju vnašamo poljubne ukaze, ki se bodo ob izvedbi migracije izvedli v bazi razdelkov. Vnesemo lahko denimo:

```
def self.up
    remove_column :paragraphs, :parent_id
    add_column :paragraphs, :parent_number, :string
end
```

S tem smo iz tabele razdelkov zbrisali stolpec z imenom `parent_id` z uporabo metode `remove_column`. S to metodo brišemo stolpce iz tabele. Nato smo dodali stolpec s številko razdelkov `parent_number` tipa niz. Vanj bomo hranili številke razdelkov tipa `"1"`, `"1.1"`, `"1.1.1"`. Uporabili smo metodo `add_column`, s katero dodajamo stolpce. V metodi `down` je dobro, če navedemo obratna ukaza, torej dodajanje stolpca `parent_id` in odvzemanje stolpca `parent_number`. S tem bomo tabelo lahko povrnili v stanje pred migracijo.
Če sedaj poženemo `rake db:migrate`, se bo na tabeli razdelkov izvedla migracija, ki smo jo napisali.

3.7.3 Analiza kode, ki se izvede ob ukazu rake routes

Usmerjanje je sistem povezovanja zahtev odjemalca HTTP z metodami krmilnikov naše aplikacije. Več o tem smo izvedeli v razdelku *Rails, Usmerjanje*. V tem razdelku bomo opisali ukaz, s katerim izpišemo definirane usmeritve naše aplikacije. Te definiramo v konfiguracijski datoteki `config/routes.rb`.

Oglejmo si, kaj se zgodi, ko poženemo ukaz `rake routes`. Koda, ki se izvede na začetku, je

popolnoma enaka kot pri ukazih `rake db:create` in `rake db:migrate`. Naložijo se datoteke, v katerih so definirana opravila vseh standardnih paketov okolja Rails. Med njimi se naloži tudi datoteka `/usr/lib/ruby/gems/1.8/gems/railties-3.0.6/lib/rails/tasks/routes.rake`. V njej je definirano sestavljeno opravilo `:routes`.

```
task :routes => :environment do

  Rails.application.reload_routes!

  all_routes = Rails.application.routes.routes

  if ENV['CONTROLLER']

       all_routes = all_routes.select{ |route|
       route.defaults[:controller] == ENV['CONTROLLER'] }

  end

  ...

  routes.each do |r|

       puts "#{r[:name].rjust(name_width)} #{r[:verb].ljust(verb_width)}
       #{r[:path].ljust(path_width)} #{r[:reqs]}"

  end

end
```

Pred blokom ukazov opravila `:routes`, se bo izvedlo opravilo `:environment`. Iz prejšnjega razdelka vemo, da se pri tem opravilu naloži datoteka `/home/andrej/ruby/diploma/config/environments/development.rb`, če le ta obstaja. V tej datoteki lahko spreminjamo konfiguracijske opcije naše aplikacije in s tem spreminjamo nastavitve, določene v datoteki `config/application.rb`.
V bloku ukazov opravila `:routes` se najprej ponovno naložijo vse usmeritve aplikacije in shranijo v spremenljivko `all_routes`. Če bi ukazu `rake routes` podali parameter `CONTROLLER=<ime_krmilnika>`, bi se v spremenljivki `all_routes` izločile vse usmeritve, ki niso povezane z določenim krmilnikom.
V naslednjem bloku `routes.each do |r|` z metodo `puts` izpišemo na standardni izhod vse parametre posameznih usmeritev. Ti parametri so ime, pot, metoda krmilnika in dodatne opcije določene usmeritve.

S tem smo zaključili spoznavanje orodja `rake`. Ogledali smo si, na kakšne načine mu lahko definiramo opravila. Preučili smo najbolj pomembni funkciji orodja – kreiranje baze podatkov in izvajanje migracij. Na koncu smo opisali še način, s katerim dobimo seznam usmeritev. Znanje pridobljeno v tem razdelku bomo uporabljali v razdelku *Rails, Model*.

3.8 Model

Razdelek opisuje del razvojnega okolja Rails, imenovan Model. V modelu se izvaja shranjevanje, preverjanje in posredovanje vseh podatkov spletne aplikacije okolja Rails. Podatki naše aplikacije se, kot vemo iz prejšnjih razdelkov, shranjujejo v bazo podatkov naše aplikacije. Model je implementiran v Rails paketu `ActiveRecord`. Razdelek bomo razdelili na dva logična sklopa.

- *Generiranje novega modela aplikacije*
  Tu si bomo ogledali postopek generiranja novega modela aplikacije. Model aplikacije generiramo za določen tip podatkov. V našem primeru bodo to razdelki diplome. Z generiranjem modela bomo ustvarili razred `Paragraph`, ki vsebuje metode za delo s tabelo razdelkov v naši bazi podatkov.
- *Metode modula ActiveRecord*
  V razdelku bomo opisali glavne metode paketa `ActiveRecord` za delo s podatki. Ogledali si bomo, kako definiramo nov razdelek in kako ga shranimo tabelo razdelkov.

Nato bomo spoznali metode za iskanje podatkov v tabeli razdelkov. Razdelek bomo zaključili z opisom metod za preverjanje podatkov in metod, ki jih lahko izvedemo pred ali po posameznih operacijah na bazi podatkov.

3.8.1 Generiranje novega modela aplikacije

V tem razdelku bomo predstavili generiranje novega modela naše aplikacije. Vemo, da bomo potrebovali bazo podatkov `diploma_development` s tabelo razdelkov. V tem razdelku se bomo posvetili pripravi razreda `Paragraph`, ki predstavlja to tabelo razdelkov. Ko je ta razred definiran, lahko poženemo ustrezen ukaz `rake db:migrate` in tabelo v bazi ustvarimo.
Najprej si oglejmo ukaz, s katerim generiramo nov model aplikacije:

```
rails generate model paragraph parent_number:string
type_paragraph:string title:string contents:text number:string
```

Beseda `paragraph`, ki sledi besedi `model`, predstavlja ime modela. Pari `ime:tip`, ki sledijo besedi `paragraph`, so imena stolpcev tabele razdelkov diplomske naloge in tip podatkov, ki jih v stolpcu hranimo. V našem primeru smo definirali vse stolpce v tabeli razdelkov. Razlaga pomena posameznih stolpcev je napisana v razdelku *Razvoj aplikacije, Določitev podatkov v bazi podatkov in generiranje modelov*.
Ko izvedemo zgornji ukaz, se zažene generator za generiranje modelov aplikacije. Glavni razred tega generatorja je razred `Rails::Generator::ModelGenerator`. Oglejmo si del vsebine razreda datoteke `/usr/lib/ruby/gems/1.8/gems/1.8/gems/railties-3.0.6/lib/rails/generators/rails/model/model_generator.rb`, kjer je razred `ModelGenerator` definiran:

```
module Rails

  module Generators

    class ModelGenerator < NamedBase #metagenerator

      hook_for :orm, :required => true
```

V zadnji vrstici kode vidimo ukaz `hook_for`. Ta metoda se uporablja za zaganjanje generatorjev, ki jih metodi `hook_for` podamo kot parametre. V našem primeru bomo zagnali generator, imenovan `orm`, s parametrom `required`.
Kratica ORM (`Object-relation mapping`) predstavlja glavno funkcionalnost paketa `ActiveRecord`. To je povezovanje podatkov v tabeli baze aplikacije z metodami objekta razreda `ActiveRecord::Base`. Z uporabo metod novo kreiranega objekta modela `Paragraph`, ki bo dedoval iz razreda `ActiveRecord::Base` bomo lahko dostopali, dodajali in spreminjali vrstice v tabeli razdelkov baze podatkov naše aplikacije.
Ob zagonu generatorja se med drugim izvede koda datoteke `/usr/lib/ruby/gems/1.8/gems/activerecord-3.0.6/lib/rails/generators/active_record/model/model_generator.rb`, katere del je:

```
def create_migration_file
      return unless options[:migration] && options[:parent].nil?
      migration_template"migration.rb","db/migrate/create_#{table_name}
      .rb"
end
def create_model_file
      template 'model.rb', File.join('app/models', class_path,
      "#{file_name}.rb")
end
```

Izvedli se bosta torej metodi `create_migration_file` in `create_model_file`. V metodi `create_migration_file` se ustvari datoteka `db/migrate/<časovna_značka>_create_paragraphs.rb`. S to datoteko bomo z uporabo

ukaza `rake db:migrate` ustvarili tabelo razdelkov naše aplikacije. Z metodo `create_model_file` dobimo datoteko `/app/models/paragraph.rb`. V tej datoteki je definiran razred `Paragraph`, razred našega modela. Tej datoteki se bomo posvetili v naslednjem razdelku.

3.8.2 Metode modula ActiveRecord

Prej smo opisali, kako generiramo razred modela tabele razdelkov `Paragraph`. Nato smo z ukazom `rake db:migrate` to tabelo razdelkov ustvarili.
Sedaj bomo spoznali metode, s katerimi urejamo vrstice v tabeli razdelkov. Vsaka vrstica v tabeli predstavlja posamezen razdelek diplomske naloge. Pri spoznavanju metod s katerimi urejamo vrstice v tabeli, bomo uporabili ukazno vrstico okolja Rails. Z uporabo ukazne vrstice bomo spoznali delovanje metod, ki jih na enak način uporablja tudi krmilnik. Ukazno vrstico bomo uporabili za prikaz delovanja posameznih metod. Za prikaz njihovega delovanja tako ne bo potrebno poganjati cele aplikacije. Uporabljali bomo metode razreda `Paragraph`, ki smo ga definirali ob kreaciji modela razdelkov. Ukazno vrstico poženemo z ukazom `rails console` iz krovnega imenika naše aplikacije `/home/andrej/ruby/diploma/`.
Večino metod, ki jih bomo uporabili, bomo kasneje videli tudi v razdelku *Rails, Krmilnik*.
Za kreiranje novega razdelka lahko uporabimo več podobnih metod, vse pa posredno ali neposredno kličejo konstruktor razreda `Paragraph`. Prikazali bomo tiste metode, ki jih bomo uporabili tudi v akcijah krmilnika.
Nov razdelek ustvarimo z ukazom:

```
> paragraph = Paragraph.new

=> #<Paragraph id: nil, parent_number: nil, type_paragraph: nil,
title: nil, contents: nil, number: nil, created_at: nil, updated_at:
nil>
```

Kot vidimo v odgovoru na ukaz, so vsi podatki trenutno prazni in jih moramo še izpolniti. Kot vemo iz razdelka *Rails, Rake* nam podatkov za `id` in `created_at` in `updated_at` ni treba vstavljati, ker se to zgodi avtomatsko, ko razdelek shranimo v tabelo razdelkov. Vpisati pa moramo ostale podatke:

```
> paragraph.type_paragraph = "chapter"

=> "chapter"

> paragraph.title = "Ruby"

=> "Ruby"

> paragraph.contents = <<stavek

" Ruby je objektno orientiran jezik

" Razvil ga je Yukihiro Matsumoto Matz

" ...

" stavek

=> "Ruby je objektno orientiran jezik\nRazvil ga je Yukihiro Matsumoto
Matz\n...\n"

> paragraph.number = "1"

=> "1"
```

Tako vnašanje je priročno, kadar spreminjamo samo določeno lastnost razdelka. Oglejmo si, kako vnesemo vse podatke v razdelek z enim ukazom:

```
> par = Paragraph.new(
* :type_paragraph => "chapter",
```

```
* :title => "Ruby",
* :contents => "Ruby je objektno orientiran jezik ...",
* :number => "1.1")

=> #<Paragraph id: nil, parent_number: nil, type_paragraph: "chapter",
title: "Ruby", contents: "Ruby je objektno orientiran jezik ...",
number: "1.1", created_at: nil, updated_at: nil>
```

Za podajanje posameznih lastnosti razdelka smo uporabili njihove simbole kot ključe slovarja. Potem, ko smo razdelek ustvarili, ga lahko shranimo v tabelo razdelkov. To storimo z ukazom:

```
> par.save

=> true
```

Razdelek smo uspešno shranili. Seveda pa želimo preveriti, kako so se lastnosti shranile v tabelo razdelkov. V ta namen bomo uporabili metode, ki jih razred `Paragraph` deduje iz razreda `ActiveRecord::Base`. Te metode se imenujejo iskalci in služijo za iskanje razdelkov v bazi razdelkov.
Za začetek poženimo ukaz:

```
> paragraphs = Paragraph.find(:all)

=> [#<Paragraph id: 1, parent_number: nil, type_paragraph: "chapter",
title: "Ruby", contents: "Ruby je objektno orientiran jezik\nRazvil ga
je Yuki...", number: "1", created_at: "2011-07-27 11:53:22", updated_at:
"2011-07-27 11:53:22">, #<Paragraph id: 2, parent_number: nil,
type_paragraph: "chapter", title: "Ruby", contents: "Ruby je objektno
orientiran jezik ...", number: "1.1", created_at: "2011-07-27 12:39:10",
updated_at: "2011-07-27 12:39:10">]
```

Ukaz `find(:all)` nam vrne tabelo vseh vnosov v tabeli razdelkov. Trenutno sta v tabeli dva razdelka.
Denimo, da poznamo naslov razdelka. Takrat lahko uporabimo ukaz:

```
> Paragraph.find_by_title("Ruby")

=> #<Paragraph id: 1, parent_number: nil, type_paragraph: "chapter",
title: "Ruby", contents: "Ruby je objektno orientiran jezik\nRazvil ga
je Yuki...", number: "1", created_at: "2011-07-27 11:53:22", updated_at:
"2011-07-27 11:53:22">
```

Uporabili smo metodo `find_by_title`, lahko pa uporabili tudi `find_by_type_paragraph`, `find_by_number` ali `find_by_id` in tako iskali po drugih lastnostih razdelka. Morda smo pričakovali, da bomo dobili dva (oba) razdelka, saj imata oba naslov "Ruby". A vsaka od teh metod pa vrne le prvi zapis, ki ustreza pogoju v parametru metode. Naštete metode se generirajo avtomatsko ob definiranju stolpcev v tabeli. Spomnimo se, da smo stolpce v tabeli razdelkov ustvarili z ukazom `rake db:migrate`.
Če želimo videti le določene lastnosti razdelkov (določene stolpce), bomo uporabili metodo `select:`

```
> Paragraph.select("id,title,number")

=> [#<Paragraph id: 1, title: "Ruby", number: "1">, #<Paragraph id: 2,
title: "Ruby", number: "1.1">]
```

Metodi smo v parametru določili, da želimo le izpis naslednjih lastnosti: `id`, `title` in `number`.
Če želimo videti tiste razdelke, ki ustrezajo določenemu pogoju (zanimajo nas podrazdelki razdelka 1), uporabimo metodo `where:`

```
> Paragraph.where('"1" < number AND number < "2"')

=> [#<Paragraph id: 2, parent_number: nil, type_paragraph: "chapter",
title: "Ruby", contents: "Ruby je objektno orientiran jezik ...",
number: "1.1", created_at: "2011-07-27 12:39:10", updated_at: "2011-07-
27 12:39:10">]
```


Metoda `where` nam pri danem pogoju `'"1" < number AND number < "2"'` vrne le prvi vnos

v tabeli razdelkov. Žal metodi `where` ne moremo določiti poljubnega nabora posameznih stolpcev pri prikazu rezultatov ukaza. Vedno vrne vse stolpce v tabeli. Lahko pa prikažemo ukaz jezika SQL, ki se izvede v ozadju pri klicu metode `where`:

```
> Paragraph.where('"1" < number AND number < "2"').to_sql

=> "SELECT \"paragraphs\".* FROM \"paragraphs\" WHERE (\"1\" < number
AND number < \"2\")"
```

Pogosto pa z metodami `where, find, …` ne moremo vedno doseči vsega, kar bi lahko opravili z ustrezno poizvedbo v jeziku SQL. Takrat uporabimo metodo `find_by_sql`. Ta metoda kot parameter dobi niz, ki vsebuje ukaz v jeziku SQL:

```
> Paragraph.find_by_sql('SELECT id,title,number FROM "paragraphs" WHERE
("1" < number AND number < "2") ')

=> [#<Paragraph id: 2, title: "Ruby", number: "1.1">]
```

Seveda obstaja še cela vrsta metod, s katerimi lahko iskanje še dodatno prilagodimo. Pri iskanju lahko iščemo tudi v več različnih tabelah, ki jih združujemo z uporabo zunanjih ključev. Vendar v naši aplikaciji povezav med različnimi tabelami ne potrebujemo. Slike, ki jih bomo hranili v tabeli slik, bomo v vsebini razdelka prikazovali na točno določenem mestu. Da bomo to lahko naredili, bomo uporabili paket `RedCloth` (glej razdelek *Razvoj aplikacije, Namestitev in uporaba paketa RedCloth.*). Vsebino razdelka bomo namreč shranjevali v formatu HTML. To je razlog, da povezave med tabelo razdelkov in tabelo slik ne potrebujemo.

Sedaj si bomo ogledali še primer metode modela, s katerimi preverjamo podatke pred vnosom razdelka v tabelo razdelkov. Na prejšnjih primerih smo opazili, da smo lahko razdelek shranili, ne glede nato, da je v tabeli razdelkov že obstajal razdelek z istim naslovom. Sedaj želimo to preprečiti. V datoteko `/app/models/paragraph.rb` razreda `Paragraph` dodamo naslednjo kodo:

```
class Paragraph < ActiveRecord::Base

  validates_uniqueness_of :title, :number

end
```

S tem smo določili, da se bo pred izvajanjem vsake metode razreda `Paragraph` izvedla metoda `validates_uniqueness_of`. Ta bo preverila, če v tabeli razdelkov morda že obstaja razdelek z naslovom ali številko razdelka enako naslovu ali številki razdelka, ki ga želimo shraniti. Preizkusimo novo metodo na primeru:

```
> par = Paragraph.new(

* :title => "Ruby"

> )

=> #<Paragraph id: nil, parent_number: nil, type_paragraph: nil, title:
"Ruby", contents: nil, number: nil, created_at: nil, updated_at: nil>

> par.save

=> false
```

Hoteli smo shraniti razdelek z naslovom "`Ruby`". Tak razdelek pa v tabeli razdelkov že obstaja. Metoda `validate_uniqueness_of` nam je preprečila shranjevanje neveljavnega razdelka. Poleg te metode poznamo še vrsto drugih metod za preverjanje podatkov:

- `validates_each` – je metoda, ki pravilnost podatkov preveri z ukazi, ki jih navedemo v bloku ukazov, podanem metodi kot parameter.
- `validates_length_of` – preveri, če so podatki ustrezne dolžine.
- `validates_presence_of` – s to funkcijo preverimo, če smo za določene stolpce določili vrednosti.

- `validates_format_of` – preveri, če je format podatkov pravi.
- `validates_numerality_of` – preveri, ali je dani podatek število.

Z zgornjimi metodami preverjamo vse podane podatke posebej, a vsakega na enak način. Včasih pa moramo preveriti tudi povezavo med posameznimi podatki. Takrat lahko napišemo lastno metodo za preverjanje podatkov pred shranjevanjem. Primer take metode bomo spoznali v razdelku *Razvoj aplikacije, Preverjanje podatkov pred vnosom v bazo podatkov*.

Ker smo v tem razdelku podatke v bazo dodajali le, da bi prikazali obnašanje metod, bomo pred zaključkom razdelka te vnose pobrisali. Uporabili bomo metodo `destroy`. Preden pa se lotimo brisanja, bomo spoznali še eno vrsto metod razreda `Paragraph`. To so metode, ki se izvajajo pred in/ali po določenih metodah razreda `Paragraph`. Te metode definiramo kot privatne metode objekta razreda `Paragraph`.

Oglejmo si primer. Pred brisanjem želimo uporabnika opozoriti, da se je lotil tveganega početja. V ta namen mu bomo poslali sporočilo z vprašanjem, ali naj nadaljujemo z brisanjem:

```
class Paragraph < ActiveRecord::Base

  validates_uniqueness_of :title, :number

  before_destroy :check_before_destroy

private

  def check_before_destroy

    puts "Želiš res pobrisati razdelek #{id} #{title} #{number}?[D/N]"

    vrni = gets

    if vrni.chomp.eql?("N")

       false

  end

end
```

Uporabniku smo s tem omogočili, da si premisli. Kadar metoda, ki se izvede pred metodo `destroy`, vrne `false`, se metoda `destroy` ne izvede:

```
> par3 = Paragraph.find(3)

=> #<Paragraph id: 3, parent_number: nil, type_paragraph: "chapter",
title: "Ruby", contents: nil, number: "1.1", created_at: "2011-07-27
15:02:02", updated_at: "2011-07-27 15:02:02">

> par3.destroy

Želiš res pobrisati razdelek 3 Ruby 1.1?[D/N]

N

=> false

> par3.destroy

Želiš res pobrisati razdelek 3 Ruby 1.1?[D/N]

D

=> #<Paragraph id: 3, parent_number: nil, type_paragraph: "chapter",
title: "Ruby", contents: nil, number: "1.1", created_at: "2011-07-27
15:02:02", updated_at: "2011-07-27 15:02:02">

> par3 = Paragraph.find(3)

ActiveRecord::RecordNotFound: Couldn't find Paragraph with ID=3
```


S tem smo končali predstavitev metod modela, ki jih bomo uporabljali v naši aplikaciji. Ogledali

smo si, kako ustvarimo nov razdelek in kako ga shranimo. Nato smo predstavili različne načine iskanja razdelkov, ki so shranjeni v tabeli razdelkov. Za konec smo si ogledali še način, kako lahko podatke preverimo, preden jih zapišemo v bazo in definirali metodo, ki se izvede pred metodo `destroy`, s katero brišemo razdelke. Metode, ki smo jih predstavili v tem razdelku, bomo srečali tudi v primerih razdelkov *Rails, Krmilnik* in v celem poglavju *Razvoj aplikacije*. Na povsem enak način, kot smo jih mi uporabili v ukazni vrstici okolja Rails, jih namreč uporablja krmilnik za pridobivanje podatkov iz baze podatkov pri sestavljanju odgovorov HTML .

3.9 Prikazovalnik

Prikazovalnik je del modela MVC, ki ga predstavlja črka V (View). Namenjen je prikazovanju spletnih strani. Spletne strani se v okolju Rails generirajo iz vzorčnih datotek. Vzorčne datoteke imajo končnico `.erb`. Del kode v teh datotekah je napisan v jeziku HTML, preostali del pa v jeziku Ruby. Vstavki, napisani v (prilagojenem) jeziku Ruby, so ločeni od preostale kode v jeziku HTML in označeni s posebnimi značkami. Del jezika Ruby, ki ga uporabljamo v vzorčnih datotekah, se imenuje Ruby ERB (Embedded Ruby).
V razdelku bomo spoznali, kako pišemo kodo vzorčnih datotek z vstavki v jeziku ERB. Spoznali bomo, kako so vzorčne datoteke naše aplikacije organizirane. Krovne vzorčne datoteke določajo strukturo vseh strani.
Paket Rails, ki iz vzorčnih datotek sestavlja odgovore HTML, se imenuje `ActionView`. Ogledali si bomo metode tega paketa, ki jih uporabljamo v krovnih vzorčnih datotekah. Zatem si bomo ogledali primer vzorčne datoteke, ki je povezana z akcijo krmilnika. Opisali bomo način vključevanja skupnih vzorčnih datotek in kako v vzorčnih datotekah dostopamo do podatkov iz modela aplikacije, ki nam jih posreduje krmilnik. Ogledali si bomo tudi poseben tip vzorčne datoteke – spletni obrazec, s katerim shranjujemo nove podatke v bazo podatkov.
Razdelek je razdeljen na naslednje logične sklope:

- *Opis uporabe standardne knjižnice ERB v vzorčnih datotekah*
  V tem razdelku si bomo ogledali preprost primer vzorčne datoteke, ki je povezana z akcijo krmilnika `index`. Spoznali bomo načine vstavljanja vstavkov, napisanih v jeziku ERB, v kodo v jeziku HTML.
- *Krovne vzorčne datoteke*
  Krovna vzorčna datoteka določa videz vseh spletnih strani v aplikaciji. V njej vključujemo kaskadne slogovne predloge CSS in knjižnice jezika Javascript. Z uporabo metode `yield` v njih prikazujemo vsebino ostalih vzorčnih datotek. V razdelku si bomo ogledali krovno vzorčno datoteko naše aplikacije `/home/andrej/ruby/diploma/app/views/layouts/application.html.erb`. Povedali bomo, kako z njeno pomočjo sestavimo odgovor HTML na zahtevo HTTP odjemalca.
- *Spletni obrazci in skupne vzorčne datoteke*
  V razdelku si bomo ogledali metode paketa `ActionView`, ki jih uporabljamo v spletnih obrazcih. Spletne obrazce uporabljamo za posredovanje podatkov aplikaciji. Ti podatki se potem shranjujejo v bazo podatkov.
  Eno glavnih pravil jezika Ruby je pravilo DRY. (`Don't repeat yourself`). Izogibati se je potrebno ponavljanju kode. Tako sta na primer vzorčni datoteki `new.html.erb` in `edit.html.erb` skorajda popolnoma enaki. Ker pa kode ne želimo ponavljati, uporabimo skupno vzorčno datoteko, ki jo potem z uporabo metode `render` (glej razdelek *Rails, Krmilnik*) prikažemo v obeh vzorčnih datotekah.

3.9.1 Opis uporabe standardne knjižnice ERB v vzorčnih datotekah

V razdelku si bomo ogledali preprost primer vzorčne datoteke in preučili uporabo vstavkov ERB. Poglejmo vzorčno datoteko `index.html.erb`, ki jo uporabljamo za prikazovanje spletne strani

kazala naše diplome:

```
<div id="vsebina">
      <div id="razdelki">
         <h2>Kazalo</h2>
         <% @paragraphs.each do |paragraph| %>
               <div id="poglavje">
                   <%= paragraph.number %><a href="/paragraphs/<%=
                   paragraph.id %>"><%= paragraph.title %></a><br />
               </div>
          <% end %>
      </div>
</div>
```

Vrstica `<% @paragraphs.each do |paragraph| %>` je primer vstavka jezika ERB v kodo HTML. Vstavek smo označili z značkama `<%` in `%>`. Označujeta kodo v jeziku Ruby, ki jo paket `ActionView` izvede ob generiranju spletne strani kazala.
Osnovno strukturo datoteke predstavlja zanka, v kateri izpisujemo določene podatke posameznih razdelkov iz tabele razdelkov. V samem telesu zanke uporabljamo tako ukaze v jeziku HTML, kot tudi vrednosti določenih spremenljivk, ki jih dobimo z izvedbo kode v vstavkih ERB. Osnovna struktura je torej:

```
<% glava zanke %>

    Ukazi v HTML in

    <%= vrednosti spremenljivk ERB %>

<% end %>
```

Vrstica `<% @paragraphs.each do |paragraph| %>` je glava zanke. V tej vrstici tudi vidimo, kako dostopamo do podatkov, ki jih krmilnik pridobi iz modela aplikacije. Kot bomo videli v razdelku *Rails, Krmilnik*, krmilnik v ta namen uporablja metode modela `Paragraph`. V našem primeru je shranil vse razdelke iz tabele razdelkov v tabelo objektov modela, imenovano `@paragraphs`. To tabelo metoda `render` avtomatsko posreduje prikazovalniku. Na tej tabeli se izvede metoda `each`. Ta bloku ukazov (ki ga zaključuje vrstica `<% end %>`) posreduje objekt posameznega razdelka, imenovan `paragraph`. Ob generiranju spletne strani bo izvedena metoda `each` povzročila, da se bodo ukazi v jeziku HTML in vrednosti spremenljivk ERB, ki sestavljajo vsebino glave izvedli za vsak objekt iz tabele objektov modela `@paragraphs`.
Na samem mestu glave zanke pa se bo izpisala prazna vrstica. Prav tako bo prazna vrstica na koncu kazala (zaradi `<% end %>`). Z značkama `<%` in `%>` namreč označujemo take bloke kode jezika Ruby, ki se izvedejo, a ne povzročijo spremembe vsebine spletne strani (razen izpisa prazne vrstice). Oglejmo si še primer vstavka ERB iz vsebine zanke. Vstavke v vsebini zanke uporabljamo za izpisovanje vrednosti spremenljivk posameznega objekta `paragraph`. V vrstici

```
<%= paragraph.number %><a href="/paragraphs/<%= paragraph.id %>"><%=
paragraph.title %></a><br />
```

smo na začetku uporabili značko `<%=`. Z značkami `<%= ... %>` smo dosegli, da se bo na mestu vstavka na spletni strani izpisala vsebina vrednosti vstavka ERB v jeziku HTML. Zgornji vstavek se za objekt `@paragraph` z vrednostmi `paragraph.number="1"`, `paragraph.id=1` in `paragraph.title="Ruby"` prevede v kodo HTML:

```
1 <a href="/paragraphs/1 %>">Ruby</a><br />
```

Znački `<%=` in `%>` uporabljamo torej za izpis vrnjene vrednosti vstavka ERB na spletni strani. Poglejmo si še dve znački, ki ju uporabljamo pri vstavkih ERB. Znački `<%` in `-%>` uporabljamo, kadar na mestu, kjer se izvaja koda jezika Ruby, ne želimo izpisa prazne vrstice. Če bi uporabili znački `<%` in `%>`, bi se na tem mestu izpisala prazna vrstica. Če pa vstavek zaključimo z `-%>`, smo preprečili izpis prazne vrstice.
Če želimo vstavek ERB uporabiti zgolj v obliki komentarja, torej preprečiti, da se pri generiranju spletne strani koda v vstavku izvede, uporabimo znački `<%#` in `%>`. Z znakom `#` smo preprečili izvajanje kode v vstavku ERB.

S tem smo spoznali način, s katerim vključujemo kodo v jeziku Ruby v vzorčne datoteke. Videli smo, da so spremenljivke, v katere v krmilniku hranimo objekte modela, dostopne tudi v prikazovalniku. Zato nam jih ni bilo treba posebej definirati ali podati metodam, ki prikazujejo spletne strani. Lastnost okolja Rails, da uporablja določena pravila poimenovanja pri implementaciji modela MVC (glej razdelek *Rails, Lastnosti*), nam omogoči, da so objekti modela avtomatično dostopni tudi v prikazovalniku.
V naslednjem razdelku si bomo pogledali vzorčno datoteko, ki določa videz vseh spletnih strani naše aplikacije
`/home/andrej/ruby/diploma/app/views/layouts/application.html.erb`.

3.9.2 Krovne vzorčne datoteke

Krovne vzorčne datoteke določajo videz vseh spletnih strani v aplikaciji. V naši aplikaciji se vzorčna spletna stran nahaja v imeniku
`/home/andrej/ruby/diploma/app/views/layouts/` in se imenuje
`application.html.erb`. Ponavadi se v aplikacijah uporablja le ta krovna datoteka. Lahko pa bi uporabili krovno datoteko le za določen krmilnik. V našem primeru se krmilnik tabele razdelkov imenuje `Paragraphs`. Če bi želeli določiti krovno vzorčno datoteko le za ta krmilnik, bi ustvarili datoteko `/home/andrej/ruby/diploma/app/views/paragraphs.html.erb`. Vanjo bi vnesli kodo, ki bi določala videz vseh spletnih strani, ki jih vračajo akcije krmilnika `Paragraphs`.
Obstaja možnost, da za posamezno spletno stran uporabimo posebno krovno vzorčno datoteko. To nam omogoča opcija `:layout` pri klicu metode `render`. Metode `render` in akcije krmilnika bomo spoznali v razdelku *Rails, Krmilnik*.
Zdaj si oglejmo vsebino datoteke `application.html.erb`, s katero bomo določili videz spletnih strani v naši aplikaciji:

```
<!DOCTYPE html>
<html>
<head>
  <title>Diploma</title>
  <%= stylesheet_link_tag :all %>
  <%= stylesheet_link_tag "slogstrani.css" %>
  <%= javascript_include_tag :defaults %>
  <%= javascript_include_tag 'mojeJS.js' %>
  <%= csrf_meta_tag %>
</head>
<body>
  <%= yield %>
</body>
</html>
```

V prvi vrstici je deklaracija tipa dokumenta. Dokument je tipa HTML. Značke `html`, `head` in `body` so standardni elementi HTML. V naslednjih vrsticah vidimo nekaj vstavkov jezika ERB. V teh vstavkih uporabljamo metodi, ki poskrbita, da se v HTML napišejo ustrezni ukazi za vključevanje kaskadnih slogovnih predlog CSS in datotek jezika Javascript.
Metoda paketa `ActionView` za vključevanje kaskadnih slogovnih predlog CSS se imenuje `stylesheet_link_tag`. Metodi lahko določimo več parametrov. Z uporabo parametra `:all` avtomatsko vključimo vse datoteke s končnico `.css`, ki se nahajajo v imeniku

/home/andrej/ruby/diploma/public/stylesheets/ (glej razdelek *Rails, Organizacija in pomen generiranih imenikov in datotek*) .
V vrstici

```
<%= stylesheet_link_tag "slogstrani.css" %>
```

smo eksplicitno vključili datoteko `slogstrani.css`. Načeloma ta vrstice ne potrebujemo, ker smo v vrstici `<%= stylesheet_link_tag :all %>` avtomatično vključili vse datoteke iz imenika `public/stylesheets/`. Datoteka `slogstrani.css` določa slog prikazovanja elementov jezika HTML, ki jih bomo definirali v naši aplikaciji.
Omenimo še tri parametre metode `stylesheet_link_tag`, ki pa jih v zgledu nismo uporabili. To so `:media=>"screen"`, `:rel=>"stylesheet"` in `:type=>"text/css"`. Parameter `:media` določa medij, ki se bo uporabil pri izpisu spletne strani. Privzeta nastavitev je zaslon (`screen`). V določenih primerih bi lahko kot medij za izpis spletnih strani namesto zaslona nastavili tudi tiskalnik. Druga dva parametra `:rel` in `:type` določata tip spletne strani povezave. Ker določamo datoteko slogovnih kaskadnih predlog, imata oba privzete vrednosti `"stylesheet"` in `"text/css"`. Ker so nam vse te privzete vrednosti ustrezale, jih v našem zgledu nismo uporabili. Ko se namreč izvede metoda `stylesheet_link_tag` v vstavku ERB se vrstica `<%= stylesheet_link_tag "slogstrani.css" %>` zamenja z vsebino značke HTML:

```
<link href="/stylesheets/slogstrani.css"  media="screen" rel="stylesheet"
type="text/css" />.
```

Metoda za vključevanje standardnih knjižnic jezika Javascript se imenuje `javascript_include_tag`. Kot prejšnji, tudi tej metodi lahko določimo več parametrov. S parametrom `:defaults` avtomatsko vključimo knjižnici jezika Javascript imenovani `prototype.js` in `scriptaculuos.js`. In čemu je potrebno uporabljati še jezik Javascript, ko pa že imamo na voljo ERB? Jezik Javascript se uporablja pri osveževanju delov spletnih strani. Spletni strežnik namreč vedno vrača le celo spletno stran. Pri določenih aplikacijah pa je zaželeno, da se deli spletnih strani osvežujejo s posodobljenimi informacijami, medtem ko preostali deli ostanejo nespremenjeni. To je mogoče doseči le z uporabo jezika Javascript. Vendar se nam ob uporabi okolja Rails jezika Javascript ni potrebno učiti, saj z uporabo določenih metod v jeziku Ruby generiramo in kličemo metode jezika Javascript. Seveda pa moramo zato vključiti primerne knjižnice jezika Javascript, kot sta denimo `prototype.js` in `scriptaculuos.js`.
V naslednji vrstici

```
<%= javascript_include_tag 'mojeJS.js' %>
```

je primer direktnega vključevanja knjižnice jezika Javascript `mojeJS.js`. Knjižnice jezika Javascript se nahajajo v imeniku aplikacije
`/home/andrej/ruby/diploma/public/javascripts/`.
Metoda `csrf_meta_tag` vrne posebno označbo, s katero preprečujemo določene spletne zlorabe (npr. vrivanje ponarejenih spletnih strani).
V elementu HTML `<body>`, ki označuje vsebino spletnih strani, opazimo vstavek ERB:

```
<%= yield %>
```

Metodo `yield` že poznamo iz razdelka *Ruby, Bloki in procedure*. Uporabljali smo jo za izvajanje bloka ukazov podanega določeni metodi razreda. V primeru krovne vzorčne datoteke z ukazom `yield` vključimo vsebino tiste vzorčne datoteke, ki predstavlja odgovor na zahtevo HTML odjemalca. Te vzorčne datoteke so povezane z akcijo krmilnika. Tako je na primer vzorčna datoteka `index.html.erb` je povezana z akcijo `index`.
Pri klicu metode `render` iz akcije `index` se najprej naloži krovna vzorčna datoteka (običajno `application.html.erb`) z glavo, v kateri so določene kaskadne slogovne predloge in knjižnice jezika Javascript. S klicem metode `yield` pa se naloži vsebina vzorčne datoteke `index.html.erb`, ki je z akcijo `index` povezana.

3.9.3 Spletni obrazci in skupne vzorčne datoteke

Spletne obrazce uporabljamo za vnašanje podatkov v bazo podatkov. Pri tem podatke, ki jih vnesemo v spletni obrazec, s klikom na gumb shranimo v bazo podatkov. V našem primeru bomo shranili nov razdelek v tabelo razdelkov z uporabo spletnega obrazca, ki ga bomo napisali v vzorčni datoteki `new.html.erb`. V vzorčno datoteko `new.html.erb,` ki mora biti v imeniku `/home/andrej/ruby/diploma/app/views/paragraphs/,` vnesemo naslednjo kodo:

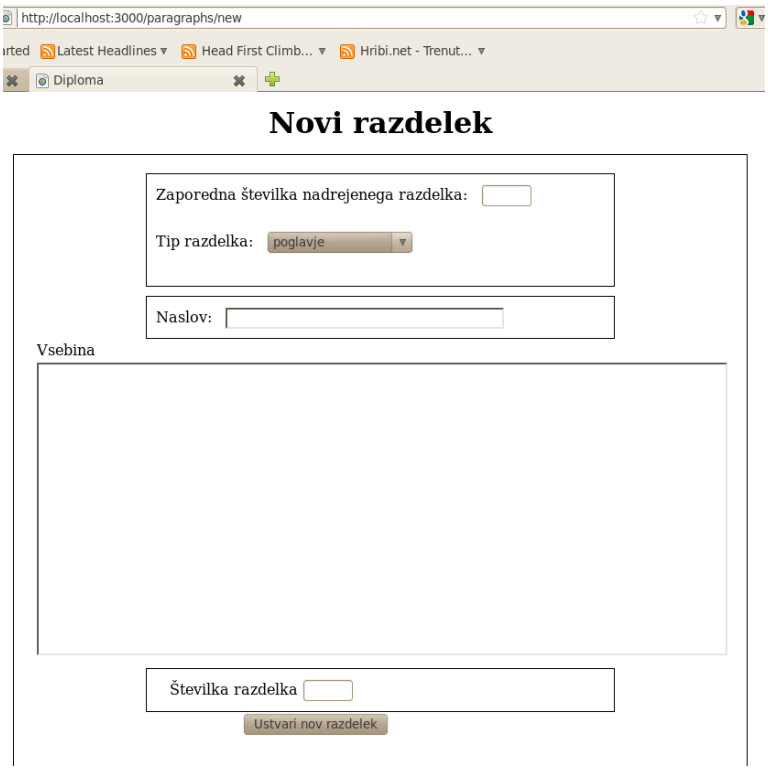
```
<%= form_for(@paragraph) do |f| %>
        <%= f.label :parent_number%>
        <%= f.text_field :parent_number %>
        <%= f.label :type_paragraph %>
        <%= f.select :type_paragraph, [["chapter","chapter"],
        ["subchapter","subchapter"],["subsubchapter","subsubchapter"],
        ["subsubsubchapter"]]%>
        <%= f.label :title %>
        <%= f.text_field :title %>
        <%= f.label :contents %><br />
        <%= f.text_area :contents %>
        <%= f.label :number %>
        <%= f.text_field :number %>
        <%= f.submit %>
<% end %>
```

V razdelku *Rails, Krmilnik* bomo videli, da se pred implicitnim klicem metode `render` najprej ustvari nov objekt razreda modela `Paragraph`. Ta objekt se shrani v spremenljivko `@paragraph`, ki je po klicu metode `render` avtomatsko dostopna v vzorčni datoteki `new.html.erb`. Ta vzorčna datoteka je primer spletnega obrazca. V jeziku HTML so spletni obrazci označeni z značko `form`. Spletni obrazec, ki bo nastal ob generiranju spletne strani `new.html` iz vzorčne datoteke `new.html.erb`, bo vseboval naslednje elemente:

- `label` – tekstovna označba določenega elementa. Služi za poimenovanje elementov, preko katerih vnašamo podatke v spletni obrazec.
- `text_field` – predstavlja vnosno vrstico, v katero vnašamo tekst. V našem primeru bomo element `text_field` uporabili za vnos naslova razdelka :`title`, številko razdelka :`number` in številko razdelka očeta :`parent_number`. Oče predstavlja razdelek, ki vsebuje naš razdelek. Na primer poglavje Ruby s številko razdelka 1 vsebuje podpoglavje Lastnosti s številko 1.2. Pri vnosu razdelka z naslovom Lastnosti moramo torej vnesti številko razdelka očeta 1 (Ruby).
- `text_area` – predstavlja ograjen blok za vnos večih odstavkov teksta. Uporabili ga bomo pri vnosu vsebine razdelka – `contents`.
- `select` – padajoč seznam, v katerem izbiramo vnaprej določene vrednosti. V našem primeru na ta način izbiramo tip razdelka (`type_paragraph`). Možne izbire pa so `chapter`, `subchapter`, `subsubchapter` in `subsubsubchapter`.
- `submit` – gumb, na katerega kliknemo po vnosu vseh podatkov za nov razdelek. Ob kliku se razdelek shrani v tabelo razdelkov.

Z metodo `form_for` v jeziku ERB povežemo objekt modela `Paragraph` z elementi spletnega obrazca, ki smo jih našteli zgoraj. Metoda `form_for` se uporablja z blokom ukazov, kateremu kot parameter poda generator spletnega obrazca `f`. Z metodami tega generatorja generiramo posamezne elemente spletnega obrazca, ki so vsebovani v elementu HTML `form`. Na primer z metodo `f.text_field` generiramo element HTML `text_field`. Metodi `form_for` podamo parameter `@paragraph` (kot vemo, je to objekt modela `Paragraph`, ki predstavlja novi razdelek). Z njim določimo imena elementov HTML, ki sestavljajo spletni obrazec. Z uporabo simbola `:title` bomo podatek o naslovu shranili pod imenom `paragraph[title]`. Po kliku na gumb `submit`, bodo podatki iz spletnega obrazca shranjeni v objektu modela `Paragraph`. Shranjene podatke bomo lahko shranili v tabelo razdelkov v naslednji akciji krmilnika. Shranjevanje si bomo ogledali v razdelku *Rails, Krmilnik*. Za boljšo predstavo si oglejmo

najprej prikaz v brskalniku:



Slika 5. Primer spletnega obrazca za novi razdelek

ter kodo spletne strani `new.html`, ki je generirana iz zgornjega spletnega obrazca:

```
<form accept-charset="UTF-8" action="/paragraphs" class="new_paragraph"
id="formbox" method="post">
<div style="margin:0;padding:0;display:inline"><input name="utf8"
type="hidden" value="&#x2713;" /><input name="authenticity_token"
type="hidden" value="tvd3js2XbERP6GWdi4P+d6KbSTTUCywxm/0KpJCe+mI=" />
</div>

<label for="paragraph_parent_number">Parent number</label>
<input id="paragraph_parent_number" name="paragraph[parent_number]"
size="30" type="text" />
<label for="paragraph_type_paragraph">Type
paragraph</label>:  
<select id="paragraph_type_paragraph" name="paragraph[type_paragraph]">
    <option value="chapter">chapter</option>
    <option value="subchapter">subchapter</option>
    <option value="subsubchapter">subsubchapter</option>
    <option value="subsubsubchapter">subsubsubchapter</option>
</select>
<label for="paragraph_title">Title</label>
<input id="paragraph_title" name="paragraph[title]" size="30"
type="text"/>
<label for="paragraph_contents">Contents</label><br />
<textarea cols="40" id="paragraph_contents" name="paragraph[contents]"
rows="20">
</textarea>
```

```
<label for="paragraph_number">Number</label>
<input id="paragraph_number" name="paragraph[number]" size="30"
type="text" />
<input id="paragraph_submit" name="commit" type="submit" value="Create
Paragraph" />
```

Če primerjamo datoteko s spletnima obrazcema za vnos novega podatka `new.html.erb` in datoteko, ki jo uporabljamo za popravljanje podatkov `edit.html.erb`, ugotovimo, da je njuna vsebina praktično enaka. Da se ponavljanju izognemo, uporabimo skupno vzorčno datoteko. Uvedemo novo vzorčno datoteko `_form.html.erb`. Vanjo bomo dali kodo spletnega obrazca, ki je omenjenima datotekama skupna. V datoteki `new.html.erb` pa bomo novo vzorčno datoteko `_form.html.erb` prikazali z uporabo metode `render`:

```
<h1>Novi razdelek</h1>
<%= render :partial => "form" %>
```

Skupne vzorčne datoteke se ločijo od ostalih po tem, da se njihovo ime vedno prične z znakom _. V datoteko `_form.html.erb` smo torej prestavili kodo našega spletnega obrazca. V posamezni vzorčni datoteki pa skupno vzorčno datoteko enostavno vključimo s tem, da metodi `render` kot parameter podamo ključ slovarja z imenom `:partial` in vrednostjo z imenom skupne vzorčne datoteke brez podčrtaja.

3.10 Krmilnik

V razdelku bomo opisali krmilnik. Krmilnik predstavlja možgane naše aplikacije. Aplikacija je narejena po modelu MVC in krmilnik (Controller) v modelu MVC predstavlja zadnja črka C. Najprej bomo opisali, kako generiramo krmilnike. V razdelku *Rails, Generiranje nove aplikacije okolja Rails* smo izvedeli, da vsi krmilniki aplikacije dedujejo iz osnovnega razreda krmilnikov aplikacije, imenovanega `ApplicationController`. Javne metode razreda vsakega krmilnika se imenujejo akcije. Opisali bomo tipične akcije krmilnika. Akcija se sproži, ko naša aplikacija sprejme zahtevo HTTP. V akciji se nato vrne odgovor HTTP. Najbolj tipičen primer odgovora je spletna stran HTML, ki jo v akcijah vračamo z metodo `render`. Posvetili se bomo tej metodi in navedli primere njene uporabe. Ogledali si bomo tudi nekaj primerov generiranja skript Javascript, ki jih uporabljamo za prikazovanje delov spletnih strani. V primerih bomo tudi videli, kako akcije krmilnika pridobivajo podatke iz modela (glej razdelek *Rails, Model*) za prikaz na vrnjenih spletnih straneh. Na koncu razdelka si bomo pogledali še nekaj pomembnih delov krmilnika, ki jih uporabljamo za posebne naloge aplikacije:

- Posredovanje napak na spletno stran razvijalca ali uporabnika
- Izvajanje nujnih opravil pred in po akciji krmilnika

Razdelek bomo razdelili na tri logične sklope:

- *Opis generiranja krmilnika in njegovih tipičnih akcij*
  Razdelek vsebuje opis načina generiranja krmilnikov. Nato bomo opisali tipične akcije, ki so definirane v razredu krmilnika.
- *Opis osnovnega razreda krmilnikov aplikacije ApplicationController in uporabe metode render*
  V tem razdelku bomo predstavili osnovni razred vseh krmilnikov aplikacije in več različnih načinov uporabe metode `render`. Uporabljamo jo za vračanje odgovorov na zahteve odjemalca, posredovane preko brskalnika. Odgovore lahko vrača na različne načine v različnih formatih HTML, XML, tekst in Javascript.
- *Opis delov krmilnika za izvajanje posebnih nalog aplikacije*
  V razdelku bomo na primerih opisali, kako posredujemo napake na spletno stran razvijalca ali uporabnika. Predstavili bomo tudi načine izvajanja nujnih opravil pred in po izvedbi posameznih akcij krmilnika.

3.10.1 Opis generiranja krmilnika in njegovih tipičnih akcij

Nov krmilnik naše aplikacije generiramo v imeniku aplikacije z ukazom:

```
rails generate controller paragraphs index show
```

Beseda `paragraphs`, ki sledi besedi `controller`, predstavlja ime krmilnika. Vse besede, ki ji sledijo, so imena akcij bodočega krmilnika. V našem primeru smo definirali akciji `index` in `show`.

Ko izvedemo ukaz `rails generate controller paragraphs index show` se zažene generator za generiranje krmilnikov aplikacije. Glavni razred tega generatorja je razred `Rails::Generator::ControllerGenerator`. Kot vsi ostali generatorji različice 3 okolja Rails tudi razred generatorja krmilnikov deduje iz razreda `Thor::Group`. To smo izvedeli v razdelku *Rails, Generiranje nove aplikacije okolja Rails*. Paket `Thor` omogoči, da se ob zagonu generatorja krmilnikov izvedejo vse metode objekta razreda `ControllerGenerator`. Oglejmo si del vsebine datoteke `controller_generator.rb`, kjer je razred `ControllerGenerator` definiran. Datoteka se nahaja v imeniku `/usr/lib/ruby/gems/1.8/gems/1.8/gems/railties-3.0.6/lib/rails/generators/rails/controller/`:

```
class ControllerGenerator < NamedBase
      ...
      def create_controller_files
            template 'controller.rb', File.join('app/controllers',
            class_path, "#{file_name}_controller.rb")
      end

      def add_routes
            actions.reverse.each do |action|
              route %{get "#{file_name}/#{action}"}
            end
      end

      hook_for :template_engine, :test_framework, :helper
end
```

Izvedli se bosta metodi `create_controller_files` in `add_routes`. Metoda `create_controller_files` nam bo v imeniku aplikacije `/home/andrej/ruby/diploma/app/controllers/` generirala datoteko `paragraphs_controller.rb`. V njej bo definiran razred `ParagraphsController`. To je razred krmilnika za delo s podatki v tabeli razdelkov naše aplikacije (glej razdelek *Rails, Model*). Razred bo vseboval dve metodi `index` in `show`, ki sta po generiranju brez vsebine. Druga metoda `add_routes` bo v konfiguracijsko datoteko usmeritev `config/routes.rb` (glej razdelek *Rails, Usmerjanje*) dodala dve usmeritvi:

```
get "paragraphs#index"

get "paragraphs#show"
```

S tem je ustvarjena povezava med zahtevami HTTP in akcijami `index` in `show` našega krmilnika. V zadnji vrstici kode vidimo ukaz `hook_for`. Ta metoda se uporablja za zaganjanje generatorjev, ki jih podamo metodi `hook_for` kot parametre. V našem primeru bomo zagnali generatorje imenovane `template_engine`, `test_framework` in `helper`.

Razred generatorja `template_engine` vsebuje metodo `copy_view_files`. V tej metodi se v imenik naše aplikacije `/home/andrej/ruby/diploma/app/views/paragraphs/` skopirata vzorčni datoteki `index.html.erb` in `show.html.erb`. (glej razdelek *Rails, Prikazovalnik.*) Uporabljata se pri generiranju odgovorov HTML ob izvedbi akcij krmilnika `index` in `show`.

Z generatorjem `test_framework` se v imeniku `/home/andrej/ruby/diploma/test/functional/` generira datoteka `paragraphs_controller_test.rb`. V njej je definiran razred

`ParagraphsControllerTest`, v katerega dodajamo metode za testiranje krmilnika `Paragraphs`.

Z generatorjem `helper` se generira datoteka `/home/andrej/ruby/diploma/test/app/helpers/users_helper.rb`. V njej je definiran modul `ParagraphsHelper`, v katerega dodajamo pomožne metode. Pomožne metode lahko uporabljamo v krmilniku in prikazovalniku. Pomožne metode bomo spoznali v razdelku *Razvoj aplikacije, Dodajanje pomožne metode razredu ParagraphsController*.

Z izvedbo generatorjev `template_engine`, `test_framework` in `helper` je generiranje krmilnika razreda `Paragraphs` končano. V naslednjem odstavku si bomo ogledali tipične akcije, ki jih ponavadi definiramo v razredu krmilnika.

Kot smo povedali na začetku razdelka, se javne metode objekta vsakega krmilnika imenujejo akcije. Akcijo krmilnik izvede, ko spletni strežnik sprejme novo zahtevo HTTP. Zahteva HTTP se z uporabo usmeritev, definiranih v `config/routes.rb`, poveže z ustrezno akcijo krmilnika. Ta praviloma generira in vrača ustrezne spletne strani. Te (če nismo definirali drugače) se nahajajo na naslovu `http://localhost:3000/paragraphs/`. Krmilnik spletne aplikacije, ki upošteva načela standarda REST (glej razdelek *Rails, Usmerjanje*), ima definirane naslednje akcije:

- `index`
  Ta akcija vrne spletno stran aplikacije `index.html`. Uporablja tabelo objektov razreda `Paragraph`, torej razreda, ki predstavlja model tabele razdelkov v bazi podatkov aplikacije (glej razdelek *Rails, Model*). Vsak objekt v tabeli objektov ustreza razdelku v tabeli razdelkov baze podatkov aplikacije. Podatki iz tabele objektov razreda `Paragraph` se prikažejo na spletni strani `index.html`, generirani iz vzorčne datoteke `index.html.erb`. Stran ponavadi uporabljamo za prikazovanje spiska ali kazala vseh elementov.
- `show`
  Akcija vrača spletno stran `<id>.html`. Tu `<id>` označuje identifikacijsko številko (ID) določenega razdelka v tabeli razdelkov baze podatkov. Akcija ustvari objekt razreda modela `Paragraph` (glej razdelek *Rails, Model*) s podatki, ki jih ima razdelek v bazi podatkov s tem ID-jem. Podatki tega razdelka se prikažejo na spletni strani `<id>.html`. Stran je generirana iz vzorčne datoteke `show.html.erb`.
- `new`
  Akcija ustvari nov objekt razreda `Paragraph`. Vrne spletni obrazec za vnos podatkov novega razdelka. Spletni obrazec se prikaže na spletni strani `new.html`, generirani iz vzorčne datoteke `new.html.erb`.
- `create`
  Akcija se izvede, ko na spletnem obrazcu spletne strani `new.html` kliknemo gumb `"Submit"`. Podatki iz spletnega obrazca se shranijo v nov objekt razreda `Paragraph`. Z uporabo metode `save` objekta `Paragraph` se novi razdelek shrani v tabelo razdelkov. Akcija `create` na koncu izvede preusmeritev na spletno stran z ID-jem novega razdelka. To stran generira že opisana akcija `show` iz vzorčne datoteke `show.html.erb`.
- `edit`
  Akcijo izvedemo za urejanje podatkov v določenem razdelku tabele razdelkov. Kot pri akciji `show` se tudi tu ustvari objekt razreda `Paragraph` z ID-jem razdelka. Vrne pa se spletna stran `<id>/edit.html` z obrazcem za urejanje podatkov v razdelku. Stran je generirana iz vzorčne datoteke `edit.html.erb`.
- `update`
  Akcija se izvede, ko na spletnem obrazcu spletne strani `edit.html` kliknemo gumb `"Submit"`. Podatki iz spletnega obrazca se shranijo v objekt razreda `Paragraph`. Z

uporabo metode `update_attributes` objekta `Paragraph` se razdelek v tabeli razdelkov uredi z novimi podatki. Akcija `update` na koncu izvede preusmeritev na spletno stran z ID-jem urejenega razdelka. To stran generira že opisana akcija `show` iz vzorčne datoteke `show.html.erb`.

- `destroy`

  Akcija se izvede, ko na spletni strani z ID-jem razdelka sledimo povezavi za brisanje razdelka. V akciji se kreira objekt `Paragraph` s podatki razdelka. Na tem objektu se pokliče metoda `destroy`, ki pobriše razdelek iz tabele razdelkov. Akcija na koncu izvede preusmeritev na spletno stran `index.html`. To stran generira že opisana akcija `index` iz vzorčne datoteke `index.html.erb`.

Spoznali smo tipične akcije krmilnika aplikacije in njihove naloge. Za vse akcije velja, da s kreiranjem objektov razreda `Paragraph` modela aplikacije pridobijo podatke o razdelkih iz tabele razdelkov baze aplikacije. Te podatke potem prikazujejo na spletnih straneh, ki se generirajo iz vzorčnih datotek prikazovalnika. Za prikazovanje spletnih strani akcije krmilnika uporabljajo metodo `render`, ki jo bomo spoznali v naslednjem razdelku.

3.10.2 Opis osnovnega razreda krmilnikov aplikacije ApplicationController in uporabe metode render

V modulu `ActionController` so vse metode in razredi, ki jih krmilniki uporabljajo. Tam je definiran tudi razred `ApplicationController`, glavni razred krmilnika naše aplikacije. Iz tega razreda dedujejo vsi krmilniki naše aplikacije, sam pa deduje iz razreda `ActionController::Base`. Ta vsebuje najpomembnejšo metodo, ki jo uporabljajo vsi krmilniki aplikacij, metodo `render`.

Ko preko brskalnika podamo zahtevo HTTP po določeni strani, metoda vrne ustrezno vsebino spletne strani. Metodo kličemo s parametrom `options`. Parameter `options` je spremenljivka tipa slovar (glej razdelek *Ruby, Tabele in slovarji*). Z njo metodi določimo tip vsebine, ki ga bo vrnila:

```
class ParagraphsController < AplicationController
def index
    render :action => 'index'
end
```

V zgornjem primeru vidimo tipično uporabo slovarja `options`. Ključ elementa v tem slovarju je simbol `:action`. Njegova vrednost je `'index'`. Torej nam bo na tem mestu metoda vrnila vsebino vzorčne datoteke `index.html.erb`. Obstaja pa tudi poenostavljena različica metode `render`. Razvili so jo, ker so ugotovili, da večina klicev v kodi uporablja simbol `:action`:

```
def index
    render 'index'
end
```

Ta klic metode `render` je pomensko enak zgornjemu, simbol `:action` je v tem klicu privzet. Zadevo pa lahko zapišemo še krajše. V primeru, da v metodi ne uporabimo stavka `render`, se ta avtomatsko izvede s parametrom `:action` z vrednostjo imena akcije. Ta je v našem primeru `index`:

```
def index
end
```

Vsi trije zapisi metode so torej enakovredni. Poudariti je potrebno, da se znotraj vsake akcije krmilnika metoda `render` lahko izvede le enkrat.

Oglejmo si še nekaj možnosti nastavitev slovarja `options`.

Simbol `:template` se uporablja podobno kot simbol `:action`. Razlika je v tem, da je vrednost simbola `:action` vedno le ime vzorčne datoteke. Ta mora biti na ustreznem mestu v imeniku `app/views/paragraphs/`. Pri uporabi simbola `:template` pa imamo več svobode, saj navedemo relativno pot do vzorčne datoteke:

```
def index
    render :template => 'app/views/paragraphs/index'
```

```
end
```

Oba simbola `:template` in `:action` predstavljata tip vsebine odgovora na zahtevo odjemalca. Med simbole, ki prav tako predstavljajo tip vsebine, spadajo:

- `:partial`

  Simbol se uporablja za vključevanje vsebine spletne strani iz posebne vzorčne datoteke. Ime te posebne vzorčne datoteke se začne z znakom _. Denimo, da imamo v naši aplikaciji v imeniku `app/views/` datoteko `_paragraph.html.erb` z naslednjo vsebino:

  ```
  <p>
  <b>To je test</b>
  </p>
  ```

  Vsebino te spletne stran želimo vključiti v kazalo aplikacije. Vemo, da se kazalo prikaže z izvedbo akcije `index`. Ker v akciji `index` kličemo metodo `render` s parametrom `index.html.erb`, moramo klic metode `render` vključiti v vzorčno datoteko. V akciji krmilnika namreč ni mogoče dvakrat poklicati metode `render`. Zato na konec datoteke `index.html.erb` dodamo:

  ```
  <%= render :partial => "paragraph" %>
  ```

  S tem pri prikazovanju kazala na dnu zagledamo še polkrepki napis "**`To je test`**". Tipično se simbol `:partial` uporablja pri vključevanju spletnih strani, ki so povezane z akcijami ostalih krmilnikov aplikacije. Lahko bi denimo prikazovali sliko iz tabele slik, pri čemer bi vključili vzorčno datoteko krmilnika `ImagesController`, ki je odgovoren za slike.

- `:xml`

  Simbol se uporablja za vračanje vsebine v formatu XML:

  ```
  def testXML

      render :xml => { :name => "Andrej"}.to_xml

    end
  ```

  V oknu brskalnika navedemo naslov `http://localhost:3000/paragraphs/testXML` in prikaže se stran z vsebino v formatu xml:

  ```
  <hash>

      <name>Andrej</name>

  </hash>
  ```

- `:json`

  Simbol se uporablja za vračanje vsebine v formatu JSON. Format JSON (`Javascript Object Notation`) se uporablja pri zapisovanju objektov jezika Javascript. Navedimo le primer:

  ```
  def testJson
      render :json => { :name => "Andrej"}.to_json
  end
  ```

- `:update`

  Simbol se uporablja za generiranje datotek s kodo v jeziku Javascript. Delovanje si oglejmo na primeru:

  ```
  def testUpd
      render :update do |page|
        page.alert " Z jezikom Javascript generirana skripta"
      end
  end
  ```

  Bločna konstanta `page` je objekt razreda `JavaScriptGenerator`. V tem primeru bomo generirali preprosto skripto, ki izvede metodo `alert` v jeziku Javascript. V oknu brskalnika se na naslovu `http://localhost:3000/paragraphs/testUpd` prikaže koda generirane skripte v jeziku Javascript:


```
try {
```

```
            alert(" Z jezikom Javascript generirana skripta");
        }
        catch (e) {
            alert('RJS error:\n\n' + e.toString());
            alert('alert(\" Z jezikom Javascript generirana
             skripta\");');
            throw e
        }
```

- `:js`

  Simbol se uporablja pri spreminjanju delov vsebine spletne strani. Tipično se uporablja za zamenjavo povezave do druge spletne strani na strani z vsebino druge spletne strani. To je možno le z uporabo jezika Javascript. Oglejmo si primer.
  V vzorčno datoteko `index.html.erb` dodamo povezavo do testne strani `paragraphs/test`:

```
<div id="moj javascript">
    <%= link_to "Zamenjaj z vsebino razdelka z id 6",
    test_paragraph_path, :remote => true %>
</div>
```

Dodajanje povezav do ostalih spletnih strani v vzorčne datoteke smo videli v poglavju *Rails, Prikazovalnik*. Uporabili smo parameter `remote => true`, kar pomeni, da se bo na tem mestu izvedla skripta Javascript. S potjo `test_paragraph_path` smo določili tudi, da povezava vodi do strani `http:/localhost:3000/paragraphs/test`. Torej se ob klicu povezave izvedla akcija krmilnika razdelkov, imenovana `test`. Zato v metodo `test` dodamo naslednje vrstice:

```
@paragraph = Paragraph.find(6)
render :js => "$('moj
    javascript').replace(#{@paragraph.contents.to_json})"
```

Akcija `test` se torej izvede po kliku na povezavo imenovano "`Zamenjaj z vsebino razdelka z id 6`" na spletni strani kazala. V vrstici `@paragraph = Paragraph.find(6)` najdemo z uporabo modela razdelek z ID-jem 6 (glej razdelek *Rails, Model*). V naslednji vrstici pa zamenjamo "spletno povezavo", ki jo predstavlja tekst "`Zamenjaj z vsebino razdelka z id 6`" z vsebino razdelka z ID-jem 6. Slednjo smo prej prevedli v format JSON, ki ga razume jezik Javascript. Z `:js` se je na strežniku izvedla generirana skripta jezika Javascript in vrnila vsebino razdelka, ki smo jo z metodo `render` prikazali v oknu brskalnika. Na spletni strani `paragraphs/test` smo torej ob kliku na povezavo "`Zamenjaj z vsebino razdelka z id 6`" zamenjali to povezavo z vsebino razdelka z ID-jem 6. Zamenjali smo torej del vsebine spletne strani. To pa lahko naredimo le z uporabo jezika Javacript. Zato smo uporabili simbol `:js` s katerim smo značko HTML "`moj javascript`" zamenjali z vsebino razdelka, podano v primernem formatu za jezik Javascript.

Poleg metode `render` bomo v aplikaciji občasno uporabljali tudi metodo `redirect_to`. Metoda `redirect_to` se uporablja za preusmerjanje z ene spletne strani na drugo. To običajno potrebujemo, kadar se kot posledica neke akcije krmilnika pojavi potreba po prikazu spletne strani, ki je povezana z drugo akcijo krmilnika. Tipični primer uporabe tega ukaza je primer, ko s pomočjo obrazca ustvarimo nek razdelek. Takrat želimo, da se potem, ko smo ga naredili, v brskalniku prikaže stran z njegovo vsebino. Kot smo omenili prej, razdelek shranimo v akciji `create`. Nato izvedemo ukaz:

```
@paragraph = Paragraph.find(params[:id])

if @paragraph.save

    redirect_to @paragraph
```

S tem smo brskalnik preusmerili na spletno stran, ki jo določa spremenljivka `@paragraph`. V tej spremenljivki je shranjen naslov novo ustvarjenega razdelka `http://localhost:3000/paragraphs/<id>`. Ta pot spletnega naslova je povezana z akcijo `show`. Torej se bo naslednja izvedla akcija `show`, ki nam bo z uporabo vzorčne datoteke `show.html.erb` prikazala spletno stran s podatki novo ustvarjenega razdelka –

http://localhost:3000/paragraphs/<id>.

S tem smo zaključili razdelek z opisom lastnosti razreda `ApplicationController` in njegove najbolj pomembne metode `render`. V naslednjem razdelku si bomo ogledali še metode, ki jih v krmilniku uporabljamo za posebne naloge.

3.10.3 Opis delov krmilnika za izvajanje posebnih nalog aplikacije

V razdelku bomo najprej prikazali del krmilnika, ki je odgovoren za prikazovanje napak. Ta del krmilnika se imenuje `Flash`. Navkljub imenu nima nič skupnega s tehnologijo Flash podjetja Adobe, ki se prav tako pogosto uporablja na spletnih straneh. `Flash` je objekt, ki je podoben slovarju. Služi za prenašanje objektov sporočil med eno zahtevo HTTP in naslednjo. Denimo, da želimo vedeti, ali je bilo vnašanje razdelka v bazo uspešno. Na spletni strani novega razdelka želimo videti sporočilo, da je bil razdelek uspešno shranjen. V akcijo `create` bomo pri klicu metode `redirect_to` dodali:

```
redirect_to @paragraph, :flash => {:notice => "Razdelek je bil uspešno
shranjen."}
```

Namesto tega bi lahko napisali tudi:

```
redirect_to @paragraph, flash[:notice]  => "Razdelek je bil uspešno
shranjen."
```

ali:

```
redirect_to @paragraph, :notice => "Razdelek je bil uspešno shranjen."
```

ali:

```
redirect_to @paragraph, flash.notice = "Razdelek je bil uspešno
shranjen."
```

Vse vrstice izvedejo isti ukaz. S tem ukazom pošljemo spletni strani, definirani v spremenljivki `@paragraph`, sporočilo, naj prikaže niz `"Razdelek je bil uspešno shranjen."`. Iz prejšnjega razdelka vemo, da se po metodi `create` po izvedbi ukaza `redirect_to` izvede akcija `show`, ki prikaže spletno stran novega razdelka. Da bi bilo na tej strani prikazano tudi naše novo sporočilo, moramo v vzorčni datoteki prikazati tudi sporočila orodja `Flash`. To storimo tako, da na začetek vzorčne datoteke `show.html.erb` dodamo naslednje vrstice:

```
<% if flash[:notice] %>
  <div class="notice">
     <%= flash[:notice] %>
  </div>
<% end %>
```

Ta koda prikaže vsa sporočila tipa `:notice` shranjena v objektu `Flash`. Prikažemo jih le, kadar taka sporočila obstajajo. Objekt `Flash` ima še eno pomembno lastnost. Sporočilo se prikaže le prvič, ko v brskalniku zamenjamo spletno stran. Če stran osvežimo, sporočilo izgine. To je pomembno, ker smo želeli le potrditev, da je naš razdelek uspešno shranjen. Sporočilo bi bilo v primeru, da si želimo spletno stran ponovno ogledati, odveč.
Simbol `:notice` (glej razdelek *Ruby, Simboli*) je rezervirana beseda, ki predstavlja tip sporočila. Ta simbol pomeni, da je bilo sporočilo navadno obvestilo. Drug tip sporočila `:warning` uporabljamo pri sporočanju opozoril. Tipe sporočila `:error`, `:failure` in `:alert` uporabljamo pri sporočanju napak.
Objekt `Flash` vsebuje tudi metodo `discard` za brisanje vseh ali določenih sporočil.
Če želimo določeno sporočilo na spletni strani prikazovati dlje časa, bomo uporabili metodo `keep`. Z uporabo te metode bo sporočilo ostalo na spletni strani tudi po osveževanju. To lahko uporabimo na primer takrat, ko je prišlo do napake, ki je nekaj časa še ne odpravimo. Pri naslednji akciji, ko se bo spet prikazala ista spletna stran s popravljenimi podatki in bo napaka

odpravljena, pa uporabimo metodo `discard` in tako sporočilo zbrišemo.

Sedaj si bomo ogledali še metode, ki jih uporabljamo pred in po izvedbah akcij krmilnika. Te metode imenujemo filtri. Filtre definiramo kot privatne metode krmilnika. S tem jih ločimo od javnih metod, ker so vse javne metode krmilnika akcije. Filtrom lahko določimo čas izvajanja. Izvajajo se lahko pred in/ali po določeni ali vseh akcijah krmilnika. Filtru s parametrom `:only` in seznamom akcij določimo le tiste akcije pred katerimi (oz. po katerih) naj se izvede. S parametrom `:except` pa izključimo iz seznama vseh akcij tiste akcije, pred katerimi (oz. po katerih) se filter ne bo izvedel.
Oglejmo si naslednji primer. Denimo, da želimo pred izvedbo akcije `index` na strani kazala prikazati sporočilo "`Začasno kazalo`". V razred `ParagraphsController` bomo dodali privatno metodo `testfilter`:

```
private

  def testfilter

     flash.notice = "Začasno kazalo"

  end
```

Definirali bomo, da naj se metoda `testfilter` izvede samo pred akcijo `index`. Na začetek razreda dodamo vrstico:

```
class ParagraphsController < ApplicationController

  before_filter :testfilter, :only => "index"
```

Ko sedaj prikazujemo kazalo, se nam sporočilo "`Začasno kazalo`" prikaže vsakič, ker se metoda `testfilter` izvede pred vsakim prikazovanjem kazala. Sporočilo bomo seveda umaknili, ker to ni tipična naloga za filtre. Tipična naloga za filter bi bila denimo izvedba prijave uporabnika pred dostopom do spletne strani.

S tem smo končali razdelek o krmilniku. Predstavili smo njegovo generiranje, njegov osnovni razred, akcije in najpomembnejše metode, ki jih uporabljamo v naši spletni aplikaciji. Videli smo, da akcije krmilnika podatke iz modela aplikacije pridobivajo z uporabo metod razreda modela `Paragraph`. To počnejo na način, ki smo ga podrobneje spoznali v razdelku *Rails, Model*. Na koncu razdelka smo si ogledali še uporabo metod razreda `Flash` za prikazovanje sporočil na spletnih straneh in uporabo filtrov, ki jih uporabljamo pred ali po izvajanju določenih akcij krmilnika.

3.11 Testiranje

V razdelku bomo opisali testiranje spletne aplikacije v okolju Rails. Ob razvoju spletne aplikacije hkrati razvijamo tudi metode s katerimi testiramo posamezne segmente testne aplikacije. Pogosto pri razvoju pri načrtovanju nove metode najprej napišemo metodo, s katero jo bomo testirali. Testno metodo potem lahko poženemo. V začetku bo seveda javljala napake metode, ki jo testiramo. S popravljanjem metode pa na koncu dosežemo, da se tudi testna metoda uspešno izvede. Ta način razvijanja se imenuje "`Unit testing`".
V okolju Rails se ob generiranju modela in krmilnika aplikacije (glej razdelka *Rails, Model* in *Rails, Krmilnik*) ustvarijo tudi razredi, ki nam služijo za dodajanje testnih metod za testiranje. Z njimi bomo testirali metode razredov naše aplikacije. V razdelku si bomo ogledali primer testne metode, s katero testiramo metode modela aplikacije. Povedali bomo, v katero datoteko metodo shranimo. Razdelek bomo zaključili z izvedbo testne metode.
V razdelku *Rails, Generiranje novega modela aplikacije* smo povedali, da se ob generiranju modela aplikacije ustvari tudi datoteka
`/home/andrej/ruby/diploma/test/unit/article_test.rb`.
Datoteka vsebuje po generiranju naslednjo kodo:

```
require 'test_helper'
class ParagraphTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  test "<ime>" do
    assert true
  end
end
```

V prvi vrstici je vključen modul `test_helper`. V njem je definiran razred `ActiveSupport::TestCase`, iz katerega deduje razred testnih metod `ParagraphTest`. Opazimo tudi vzorec, s katerim bomo dodajali testne metode. Začne se s ključno besedo `test`, s katero definiramo testno metodo. Metodi je podan parameter "`<ime>`", s katerim testno metodo poimenujemo. Testiranje pa izvedemo v bloku ukazov `do...end`.
V našem primeru bomo napisali novo metodo `validate_number_format` razreda modela `Paragraph`. Metoda bo služila za preverjanje formata številke razdelka `number` (in številke očeta razdelka `parent_number`). Želimo, da bi bile te številke oblike "`1`", "`1.1`", "`1.1.1`" ali "`1.1.1.1`". V razred `Paragraph` dodamo metodo:

```
def validate_number_format
    [parent_number, number].each do |num|
       t = num.split(/\./)
       return false if (t.size < 1) or (t.size > 4)
       t.each do |num|
          return false if (num.to_i == 0) and (num != "0")
       end
    end
    true
  end
```

Metoda preverja pravilnost formata vnešene številke razdelka `number` in številke očeta razdelka `parent_number`. Vsaka številka razdelka je lahko sestavljena iz najmanj enega in največ štirih števil, ki so med seboj ločena z piko. Če je ta pogoj izpolnjen, sta spremenljivki napisani v sprejemljivem formatu.
Za testiranje te nove metode modela v razred `ParagraphTest` dodamo naslednji dve metodi:

```
def setup
    @par = Paragraph.new( :title => "Ruby",
                          :contents => "Tekst ...",
                          :parent_number => "",
                          :number => "1",
                          :type_paragraph => "chapter"
                         )
end
test "validate_number_format" do
    assert @par.validate_number_format
end
```

Z metodo `setup` smo si pripravili začetne podatke. Ustvarili smo nov razdelek in ga shranili v spremenljivko `@par`. V metodi `test` pa bomo z makrojem `assert`, ki preverja vrnjeno vrednost metode (`true` ali `false`) preverili, če metoda deluje pravilno.
Za izvedbo metode `test "validate_number_format"` poženemo ukaz:

```
rake test:units --trace
```

in dobimo naslednji izpis:

```
Started

..F

Finished in 0.568174 seconds.


  1) Failure:

test_validate_number_format(ParagraphTest)
[/test/unit/paragraph_test.rb:13]:

<false> is not true.
```

Očitno smo v metodi naredili napako. Spremenljivka `parent_number` v naših testnih podatkih ne ustreza kriterijem metode `validate_number_format`. Vendar so podatki pravilni. Za poglavja (`chapter`) mora biti spremenljivka `parent_number` brez vsebine. Metodo `validate_number_format` ustrezno popravimo:

```
def validate_number_format

    tabela = [number]

    tabela << parent_number unless parent_number.empty?

    tabela.each do |num|

       t = num.split(/\./)

       return false if (t.size < 1) or (t.size > 4)

       t.each do |num|

          return false if (num.to_i == 0) and (num != "0")

       end

    end

    true

  end
```

in še enkrat poženemo ukaz `rake test:units --trace`

```
Started

...

Finished in 0.355112 seconds.


3 tests, 3 assertions, 0 failures, 0 errors
```

Metoda `validate_number_format` se sedaj obnaša v skladu z našimi pričakovanji in v razredu modela `Paragraph` jo lahko uvrstimo med metode, s katerimi bomo preverjali ustreznost podatkov pred vnosom v tabelo razdelkov:

```
class Paragraph < ActiveRecord::Base

  validates_uniqueness_of :number

  validates :validate_number_format
```

Na kratko smo si ogledali postopek testiranja in kako z njegovo pomočjo odkrivamo napake v metodah, ki jih dodajamo. Omenimo še, da lahko v testnem delu okolja Rails poleg unit testov, ki jih uporabljamo za preverjanje delovanja posameznih metod, izvajamo tudi integracijske teste, s katerimi preverjamo, kako so metode in razredi med seboj povezani. Poleg vgrajenega testnega okolja pa za testiranje obstaja še vrsta dodatnih paketov Rails. Nekateri med njimi so med programerji bolj priljubljeni kot vgrajeno testno okolje.

4 Razvoj aplikacije

4.1 Opis projekta

Tip aplikacije, ki jo razvijamo, se imenuje spletna knjiga. Aplikacija bo hranila in prikazovala razdelke diplomske naloge. To pomeni, da bomo razdelke iz dokumenta `diploma.odt` lahko shranili v bazo podatkov naše aplikacije in prikazovali na spletnih straneh naše aplikacije. Posamezne razdelke v diplomi bomo lahko dodajali, spreminjali in urejali. Ves tekst in slike bodo shranjene v bazi podatkov tipa MySQL, imenovani `diploma_development` v tabelah razdelki in slike. Stolpce v obeh tabelah bomo natančno opisali v naslednjem razdelku. Tabela razdelkov bo vsebovala naslednje stolpce:

- identifikacijsko številko razdelka (`id`)
- naslov razdelka
- vsebino razdelka
- tip razdelka
- številko razdelka očeta.

Tabela slik bo vsebovala naslednje stolpce:

- naslov slike
- vsebino slikovne datoteke.

Slike bomo v celoti shranjevali v bazo podatkov in jih prikazovali znotraj vsebine razdelkov. Aplikacija bo omogočala prikazovanje podatkov v slovenskem jeziku. Na začetni strani aplikacije bo naslov diplome, avtor in naslov fakultete ter povezavi do kazala in vsebine. Kazalo bo vsebovalo urejen seznam razdelkov, vsebina pa urejen seznam razdelkov z vsebino.

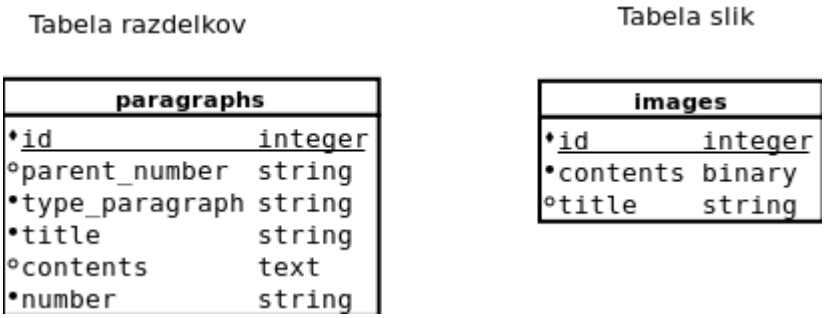
Naslov vsakega razdelka v kazalu bo predstavljala povezava do spletne strani razdelka. Na strani razdelka bomo razdelek lahko uredili ali pobrisali. Na spletni strani kazala bo tudi povezava za kreiranje novega razdelka v tabeli razdelkov. Aplikacija bo uporabljala privzet spletni strežnik Webrick ter bazo podatkov tipa MySQL.

Slika 6. Kazalo bodoče aplikacije

V razdelku *Rails, Generiranje nove aplikacije okolja Rails* smo opisali generiranje okostja nove aplikacije `diploma`. Odločili smo se, da bomo uporabljali vrsto relacijske baze podatkov tipa MySQL in privzeti spletni strežnik tipa WEBrick. Kako nastavimo aplikacijo, da uporablja sistem MySQL, smo izvedeli v poglavju *Rails, Konfiguracija baze podatkov*. Ker smo se odločili, da bomo v aplikaciji na spletnih straneh prikazovali razdelke iz naše diplome, moramo sedaj definirati tabelo v bazi podatkov, v katerih bomo hranili podatke o razdelkih.

4.2 Določitev podatkov v bazi podatkov in generiranje modelov

Najprej bomo opisali podatke, ki se bodo shranjevali v bazi podatkov. Za vsako tabelo bomo nato generirali nov objekt razreda `ActiveRecord`. Razred `ActiveRecord` je krovni modul razredov dela aplikacije, ki se imenuje Model. Spoznali smo ga v razdelku *Rails, Model*. Model v modelu MVC predstavlja M in je odgovoren za delo z razredom za posamezno tabelo.
V naši aplikaciji bomo uporabili dve tabeli. Prva je tabela razdelkov, ki vsebuje besedilo posameznega razdelka, druga pa tabela, v kateri bomo shranjevali slike iz diplome. Tabeli sta predstavljeni na naslednji sliki:



Slika 7. Baza podatkov diploma_development

Na zgornji sliki je določena organizacija podatkov v bazi podatkov naše aplikacije. Kot vidimo, je ključna tabela razdelkov. V tej tabeli so podatki vseh razdelkov v aplikaciji `diploma`. Vsak vnos ima naslednje lastnosti, ki predstavljajo stolpce tabele razdelkov:

- `id` (tipa integer) – primarni ključ tabele razdelkov.
- `parent_number` (tipa string) – id očeta razdelka, za poglavja je `null`.
- `type_paragraph` (tipa string) – dovoljene vrednosti tipa razdelka so "`chapter`", "`subchapter`", "`subsubchapter`", ali "`subsubsubchapter`".
- `title` (tipa string) – naslov razdelka.
- `contents` (tipa tekst) – to je vsebina razdelka. Vsebino razdelka bomo shranjevali v formatu HTML. V ta namen bomo namestili v našo aplikacijo dodaten vtičnik (plug-in) `RedCloth`. Tako bomo v razdelkih lahko prikazovali slike, povezave, naslove, sezname in ostale tipe podatkov, ki jih uporabljamo pri pisanju naloge v urejevalniku teksta OpenOffice.
- `number` (tipa string) – to je številka pred razdelkom, in je tipa "`1`", "`1.2`", "`1.3.1`","`1.4.2.3`".

Tabela slik je pomožna tabela. Uporabljamo jo za shranjevanje slik, ki jih prikazujemo ob tekstu diplome. Vsaka slika ima naslednje lastnosti, ki predstavljajo stolpce tabele slik:

- `id` (tipa integer) – primarni ključ tabele slik.
- `contents` (tipa binary) – vsebina slikovne datoteke. Slikovno datoteko hranimo v bazi podatkov v celoti in jo od tam tudi prikazujemo.
- `title` (tipa string) – predstavlja naslov slike.

Pri prikazovanju vsebine razdelka, ki vsebuje tudi sliko iz tabele slik bomo sliko z uporabo vtičnika `RedCloth` lahko vključili na pravo mesto v vsebini razdelka. Spletni naslov slike z določenim ID-jem bomo vpisali na pravem mestu v vsebini razdelka.

Sedaj si bomo ogledali generiranje modelov za tabeli razdelkov in slik. Funkcije in lastnosti modela smo spoznali v razdelku *`Rails, Model`*.
Za generiranje modela tabele razdelkov poženemo ukaz:

```
rails generate model paragraph
    parent_number:string
    type_paragraph:string
    title:string
    contents:text
    number:string
```

S tem smo naredili določene spremembe. Te so se zapisale v datoteko migracij `<timestamp>_create_paragraphs.rb`, zato tabelo razdelkov za model `Paragraph` ustvarimo z ukazom `rake db:migrate` (glej razdelek `Rails,Rake)`.
Za generiranje modela tabele slik pa poženemo ukaz:


```
rails generate model image title:string
```

Ker želimo v bazi podatkov ustvariti tabelo slik, moramo pognati novo migracijo, ki nam bo ustvarila tabelo slik. To storimo z ukazom `rake db:migrate`.
S tem smo ustvarili modela `Paragraph` in `Image`. Z njima bomo dostopali do podatkov razdelkov in slik, ki jih bomo hranili v bazi podatkov. Vse tabele so sedaj prazne. Podatke bomo dodajali preko spletnih strani.
S tem je prva stopnja, v kateri smo določili, kateri podatki sestavljajo našo aplikacije in ustvarili modele za delo, z njimi končana. Če želimo dodajati podatke preko spletnih strani, moramo najprej generirati krmilnik (glej razdelek *Rails, Krmilnik*). Temu se bomo posvetili v naslednjem razdelku. Napisali bomo kodo, ki se bo izvedla pri posameznih akcijah krmilnika. Pripravili pa bomo tudi vzorčne datoteke akcij krmilnika, ki jih bomo uporabljali za prikaz podatkov na spletnih straneh.

4.3 Priprava akcij krmilnika in vzorčnih datotek prikazovalnika

Razdelek opisuje generiranje krmilnika za tabelo razdelkov in njegove akcije. Lastnosti krmilnika smo spoznali v razdelku *Rails, Krmilnik*. V povezavi z vsako akcijo krmilnika je potrebno narediti tudi vzorčno datoteko, prileţno akciji krmilnika.
Začeli bomo s krmilnikom za tabelo razdelkov `ParagraphsController`. Generirali ga bomo in določili njegove akcije. Stadardne akcije krmilnika bodo:

- `index` – kazalo
- `show` – prikaz razdelka
- `new` – novi razdelek

Da bi ustvarili krmilnik s temi akcijami, bomo pognali:

```
rails generate controller paragraphs index show new
```

Kaj se zgodi pri generiranju krmilnika, smo videli v razdelku *Rails, Opis generiranja krmilnika in njegovih tipičnih akcij*. Po generiranju krmilnika želimo ustvariti prvi razdelek v naši aplikaciji. V ta namen bomo potrebovali akcijo `new`. V generirano datoteko krmilnika `app/controllers/paragraphs_controller.rb` vnesemo:

```
def new
     @paragraph = Paragraph.new
end
```

S tem smo ustvarili objekt razreda modela `Paragraph` in ga shranili v spremenljivko `@paragraph` objekta razreda krmilnika `ParagraphsController`. Akcije so specifične po tem, da se znotraj vsake akcije krmilnika na podlagi ustrezne vzorčne datatoteke generira tudi spletna stran. V našem primeru je vzorčna datoteka `app/views/paragraphs/new.html.erb`. Poglejmo si še to datoteko. V njej je le informacija o akciji krmilnika in lokaciji datoteke. Mi pa bi želeli dobiti spletni obrazec, s katerim bomo vnašali podatke v našo tabelo razdelkov. Zato vnesimo v datoteko naslednjo vsebino:

```
<h1>New paragraph</h1>
<%= form_for(@paragraph) do |f| %>
    <div class='field'>
       <%= f.label :parent_number %><br />
       <%= f.text_field :parent_number %>
    </div>
    ...
    <div class='field'>
       <%= f.label :number %><br />
       <%= f.text_field :number %>
```

```
    </div>
<% end %>
```

Če sedaj v brskalniku vpišemo naslov `http://localhost:3000/paragraphs/new` dobimo sliko spletnega obrazca za vnašanje podatkov. Opazimo, da manjka gumb za vnos podatkov v tabelo razdelkov. S tem gumbom bo povezana akcija `create`, ki se bo izvedla na zahtevo HTTP tipa POST (glej razdelek *Rails, Usmerjanje*). Ker bomo vnašali podatke v tabelo, moramo dodati usmeritev v datoteko `config/routes.rb` , ustrezno akcijo `create` v krmilnik in seveda gumb v naš spletni obrazec vzorčne datoteke `new.html.erb`:

config/routes.rb

```
...
match 'paragraphs(.:format)' => 'paragraphs#index', :as =>
:paragraphs, :via => :get
match 'paragraphs(.:format)' => 'paragraphs#create', :via => :post
...
```

app/controllers/paragraphs_controller.rb

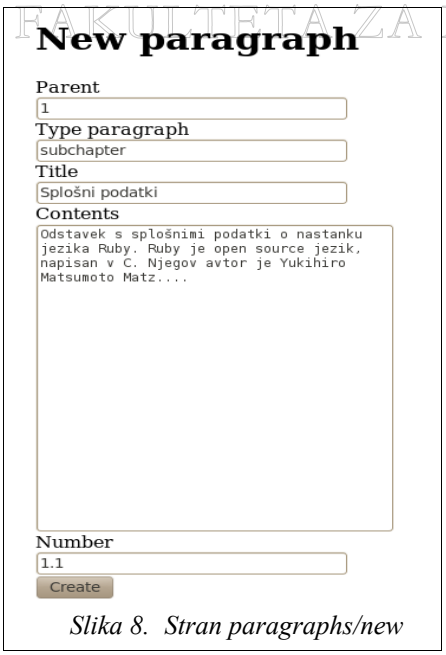
```
...
def create
   @paragraph = Paragraph.new(params[:chapter])
    @paragraph.save
   redirect_to "/paragraphs/#{@paragraph.id}"
end
...
```

app/views/paragraphs/new.html.erb

```
    ...
    <div class='actions'>
       <%= f.submit "Create" %>
    </div>
<% end %>
```

Spletna stran `http://localhost:3000/paragraphs/new` je sedaj videti takole:

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO



Slika 8. Stran paragraphs/new

Vendar pri pritisku gumba `Create` pride do napake. Po analizi ugotovimo, da je bil razdelek vnešen v bazo podatkov, zataknilo pa se je pri prikazovanju rezultatov. Zakaj? Po izvedeni akciji `create` moramo vrniti spletno stran rezultata. Za to pa nimamo narejene ustrezne vzorčne datoteke. Njena vsebina bo namenjena prikazovanju podatkov posameznega razdelka. Manjka nam tudi vsebina akcije krmilnika, metode `show`. Spremembe v kodi, ki so potrebne, so sledeče:

config/routes.rb:

```
...
match 'paragraphs/:id(.:format)' => 'paragraphs#show', :as =>
:paragraph, :via => :get
...
```

app/controllers/paragraphs_controller.rb:

```
...
def show
     @paragraph = Paragraph.find(params[:id])
  end
...
```

app/views/paragraphs/show.html.erb:

```
<h1><%= @paragraph.number %>  <%= @paragraph.title %> </h1>
<p>
   <%= @paragraph.contents %>
</p>
```

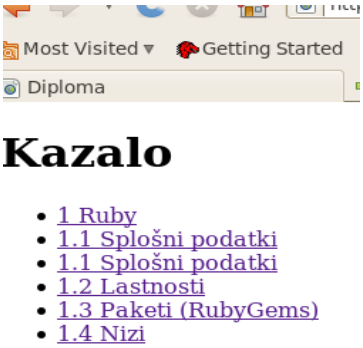
Rezultat se sedaj prikaže na spletni strani `http://localhost:3000/paragraphs/<id>`, kjer je `<id>` seveda nadomeščen z ustrezno številko:

Slika 9. Prikaz razdelka

Želimo pa si urediti še kazalo naše diplome. Kazalo naše aplikacije bo predstavljala spletna stran `http://localhost/paragraphs/index.html`. Da bi prikazovala urejen seznam razdelkov, moramo pripraviti vzorčno datoteko `app/views/paragraphs/index.html.erb` in akcijo krmilnika `index`:

```
app/views/paragraphs/index.html.erb:

<h1>Kazalo</h1>

<ul>

  <% @paragraphs.each do |paragraph| %>

    <li><a href="/paragraphs/<%= paragraph.id %>">

        <%= paragraph.number %>  <%= paragraph.title %></a>

    </li>

  <% end %>

</ul>

app/controllers/paragraphs_controller.rb

...

def index

     @paragraphs = Paragraph.find(:all)

end
```

Sedaj naša aplikacija na strani `http://localhost/paragraphs/index.html` že prikazuje kazalo.



Slika 10. Kazalo

Manjkata nam še možnosti za urejanje in brisanje podatkov. Za to bosta odgovorna akciji krmilnika, imenovani `edit` in `destroy`. V datoteko krmilnika `app/controllers/paragraphs_controller.rb` dodamo nasednjo kodo:

```
def edit
  @paragraph  = Paragraph.find(params[:id])
end
def update
  @paragraph  = Paragraph.find(params[:id])
  @paragraph.update_attributes(params[:paragraph])
  redirect_to(@paragraph)
end
def destroy
  @paragraph = Paragraph.find(params[:id])
  @paragraph.destroy
  redirect_to("/paragraphs/")
end
```

Dodali smo tudi akcijo `update`. To je akcija, ki se bo izvedla ob zahtevi HTTP za spreminjanje podatkov razdelka v tabeli razdelkov. Ta zahteva HTTP se bo izvedla ob pritisku na gumb "`Update`" na spletni strani obrazca za urejanje razdelka. Seveda moramo prej dodati primerne usmeritve v datoteko `config/routes.rb` ter napisati vzorčno datoteko `edit.html.erb` z obrazcem za urejanje razdelka:

`config/routes.rb:`

```
match 'paragraphs/:id/edit(.:format)' => 'paragraphs#edit', :as =>
:edit_paragraph , :via => :get
match 'paragraphs/:id(.:format)' => 'paragraphs#show', :as =>
:paragraph, :via => :get
match 'paragraphs/:id(.:format)' => 'paragraphs#update', :via => :put
match 'paragraphs/:id(.:format)' => 'paragraphs#destroy', :via =>
:delete
```

`edit.html.erb`

```
<h1>Edit paragraph</h1>
<%= form_for(@paragraph) do |f| %>
    <div class='field'>
       <%= f.label :parent_number %><br />
       <%= f.text_field :parent_number %>
    </div>
    ...
    <div class='field'>
       <%= f.label :number %><br />
       <%= f.text_field :number %>
```

```
    </div>
    <div class='actions'>
      <%= f.submit "Update" %>
    </div>
<% end %>
```

Videz naše strani je po teh spremembah ostal enak. To pa zato, ker do strani za urejanje in brisanje razdelkov nimamo povezav. Dodajmo povezavo za kreiranje novega razdelka na kazalo, ter povezavi za urejanje in brisanje na povezavo prikaza razdelka samega:

app/views/paragraphs/show.html.erb:

```
 <table>
<tr>
<td><%= link_to "Edit paragraph", edit_paragraph_path(@paragraph)
%></td>
<td><%= link_to "Delete paragraph", @paragraph, :confirm => 'Are you
sure?', :method => :delete %></td>
<td><%= link_to("Back",paragraphs_path) %></td>
</tr>
<table>
```

app/views/paragraphs/index.html.erb:

```
...
<%= link_to "New paragraph", new_paragraph_path %>
```

S tem smo dosegli osnovno funkcionalnost naše spletne aplikacije diplome. V naši spletni aplikaciji lahko dodajamo, spreminjamo in prikazujemo razdelke v diplomi. Seveda pa obstaja še veliko stvari, ki jim moramo urediti. Med njimi so videz strani, pravilno prikazovanje vsebine teksta razdelka (možnost izbire pisave, vstavljanja slik, zamikanja) in prevod vsega angleškega teksta na spletnih straneh v slovenščino. Vsemu temu se bomo posvetili v naslednjih razdelkih.

4.4 Pravilno prikazovanje podatkov v aplikaciji diploma

V prejšnjem razdelku smo implementirali glavni del naše aplikacije. Seveda je veliko stvari potrebno popraviti. V tem razdelku se bomo posvetili tem popravkom. Izdelali bomo pomožno metodo, ki nam bo uredila vrstni red povezav do razdelkov v kazalu naše aplikacije. Pri dodajanju novega razdelka moramo namreč poskrbeti, da se razdelki v tabeli razdelkov ustrezno preštevilčijo.
Da bomo vsebino v razdelkih lahko prikazovali v pravem formatu, bomo namestili paket `RedCloth`. Za konec bomo dodali še funkcijo za preverjanje pravilnosti vnešenih podatkov.
Razdelek je razdeljen na tri podrazdelke:

- *Dodajanje pomožne metode razredu ParagraphsController.*
  V razdelku bomo opisali dodano pomožno metodo za pravilno prikazovanje kazala.
- *Namestitev in uporaba paketa RedCloth*.
  V razdelku bomo opisali namestitev paketa `RedCloth`, ter način uporabe pri vnašanju in prikazovanju vsebine naših razdelkov.
- *Preverjanje podatkov pred vnosom v bazo podatkov*
  Prikazali bomo metodo modela `Paragraph.validate`, ki bo preverjala podatke o razdelkih pred vnosom razdelka v bazo podatkov.

4.4.1 Dodajanje pomožne metode razredu ParagraphsController

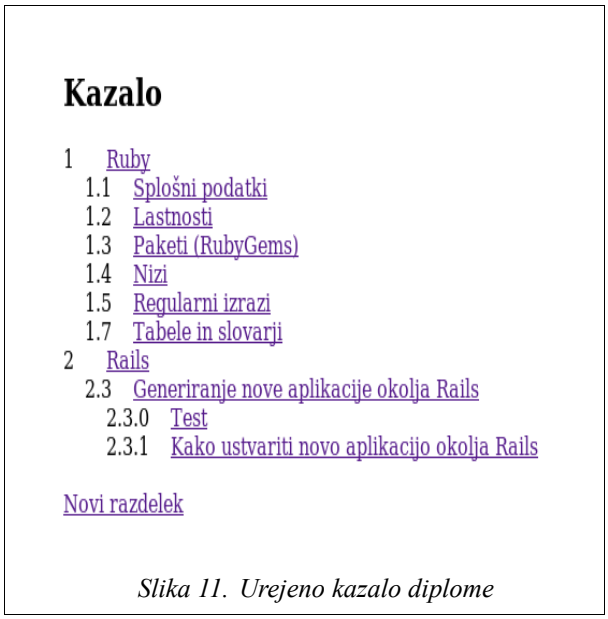
Pomožne metode smo spoznali v razdelku *Rails, Krmilnik*. Sedaj pa si bomo ogledali implementacijo metode `sort_by_number`. To bomo dodali pomožnem modulu krmilnika `ParagrahsController`, imenovanemu `ParagraphsHelper`:

```
def sort_by_number(paragraphs)
...
# shrani oznake vseh razdelkov v tabelo numbers
paragraphs.each do |paragraph|
   numbers << conv_number_to_array(paragraph.number)
end
# sortiraj tabelo numbers in shrani v sorted_numbers
sorted_numbers = numbers.sort
sorted_numbers = conv_array_to_numbers(sorted_numbers)
sorted_numbers = numbers.sort
# za vsak number najdi prileţni člen tabele razdelkov
# in ga dodaj v sorted_paragraphs
sorted_numbers.each do |num|
...
      if paragraph.number == num
         sorted_paragraphs << paragraph
         break
      end
 ...
# vrni sorted_paragraphs
return sorted_paragraphs
```

Metoda uredi vnose v tabeli `paragraphs`, ki jo dobimo iz modela `Paragraph`. Z metodo bomo vsak razdelek lahko vstavili na pravo mesto v tabeli razdelkov. Metode pomožnih modulov so, kot vemo, dostopne v vseh datotekah prikazovalnika (glej razdelek *Rails,Prikazovalnik*). Našo metodo `sort_by_number` bomo poklicali v datoteki `index.html.erb`:

```
...
<% sorted_paragraphs = sort_by_number @paragraphs %>
  <% sorted_paragraphs.each do |paragraph| %>
...
```

S tem smo seznam razdelkov uredili, preden ga prikažemo na spletni strani:



Slika 11. Urejeno kazalo diplome

4.4.2 Namestitev in uporaba paketa RedCloth.

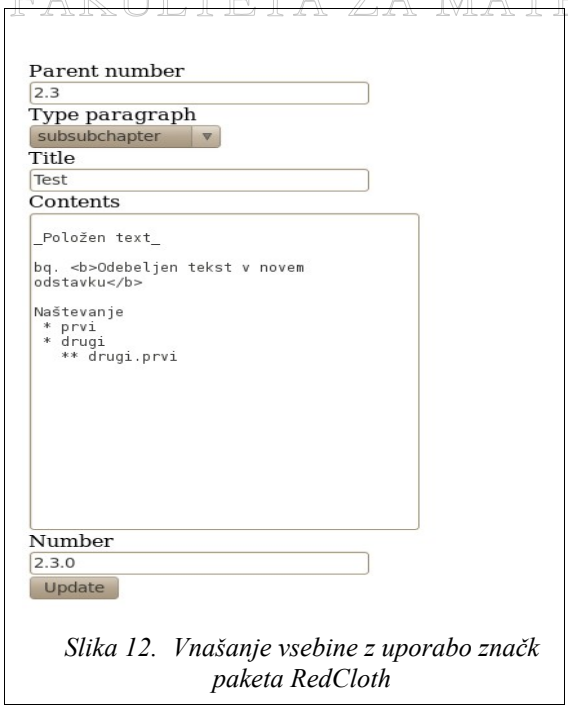
V tem razdelku si bomo ogledali namestitev in uporabo paketa `RedCloth`. Paket nam omogoča prevajanje teksta besedila v jezik HTML. V paketu `RedCloth` so definirane značke (`tag`), ki jih uporabljamo za označitev posameznih elementov teksta kot delov napisanih v jeziku HTML. Za več informacij o paketu `RedCloth` glej *Literatura in viri*.
Namestitev paketa `RedCloth` izvedemo z ukazom:

```
~/ruby/diploma# gem install RedCloth
```

Podrobnosti o paketih jezika Ruby, vrstah paketov Ruby ter njihovih namestitvah so opisane v razdelku *Ruby, Paketi (RubyGems)*.
Potem, ko je paket `RedCloth` nameščen, ga lahko uporabimo v naši aplikaciji za prikaz vsebine razdelka v jeziku HTML. V ta namen bomo v vzorčni datoteki `views/paragraphs/show.html.erb` spremenili kodo za prikazovanje vsebine razdelka v

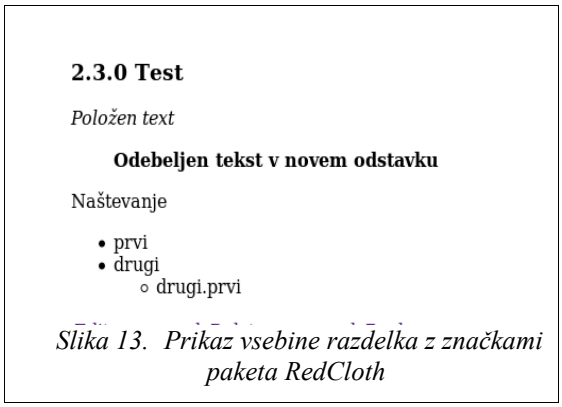
```
<div id="contents">
    <%= raw RedCloth.new(@paragraph.contents).to_html %>
</div>
```

Vsebino, ki smo jo dobili iz stolpca `contents` tabele razdelkov, podamo novemu objektu razreda `RedCloth`. Značke paketa `RedCloth`, ki so uporabljene v vsebini razdelka prevedemo v jezik HTML z uporabo metode `to_html`. Z ukazom `raw` pa dosežemo, da bomo v tem delu spletne strani lahko prikazovali besedilo v jeziku HTML.
Oglejmo si še primer, kako vnašamo nove razdelke z uporabo značk paketa `RedCloth`:



Slika 12. Vnašanje vsebine z uporabo značk paketa RedCloth

Vidimo, da poleg značk paketa `RedCloth` lahko uporabimo tudi značke jezika HTML. Oglejmo si še, kako se zgornji razdelek prikaže na spletni strani:



Slika 13. Prikaz vsebine razdelka z značkami paketa RedCloth

S tem smo dodali možnost za hranjenje in prikaz besedila razdelkov v diplomi. V naslednjem razdelku si bomo ogledali, kako preverimo podatke preden jih shranimo v tabelo razdelkov.

4.4.3 Preverjanje podatkov pred vnosom v bazo podatkov

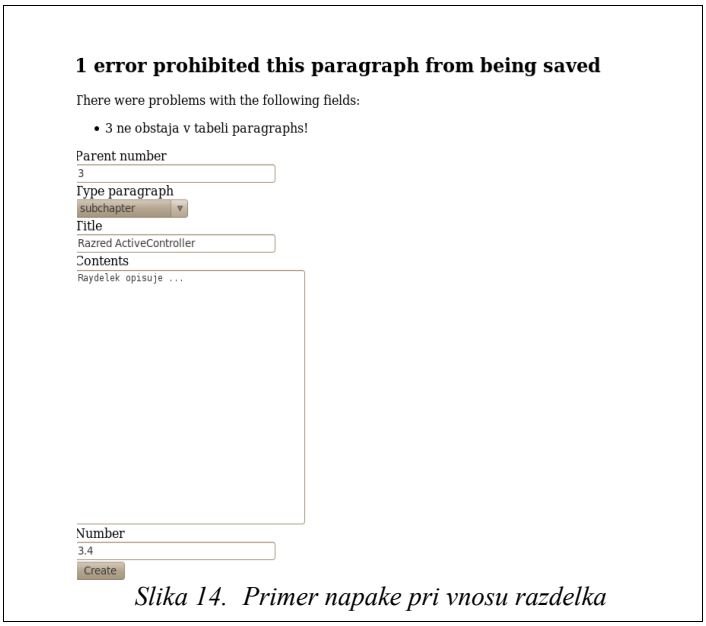
Preverjanje podatkov v aplikaciji se izvaja pred shranjevanjem, zato se vedno opravlja v modelu aplikacije. Načine preverjanja smo spoznali v razdelku *Rails, Model*. V našem primeru želimo v model razdelkov dodati metodo za preverjanje pravilnosti številk razdelkov. Številke razdelkov so nizi tipa `"1.1"`, `"1"`, `"1.1.1"`. Vsak razdelek, z izjemo poglavij, ima tudi razdelek očeta, ki mu pripada. Pravilnost podatkov bomo preverjali tako za razdelek in očeta razdelka pri vnosu novega kot pri popravku starega razdelka. Metoda, s katero bomo preverjali relacije med razdelki, se imenuje `validate`. Dodamo jo v razred modela `Paragraph`.

```
def validate
  # preveri, če ime podrazdelek očeta
  if (parent_number == "")
```

```
            if (type_paragraph != "chapter")
               errors.add(:base,"Vsako podpoglavje mora imeti
                                 določen parent_number")
               return false;
            end
         else
            # preveri, če v tabeli obstaja oče podrazdelka
            checkparagraphs = Paragraph.find(:all, :conditions =>
                                              { :number => parent_number })
            if checkparagraphs.size != 1
                errors.add(:base, "#{parent_number} ne
                                   obstaja v tabeli paragraphs!")
                return false
            end
         end
         # preveri, če oznaka številke, oznaka številke očeta,
         # tip razdelka ustrezajo kriterijem
         ret = check_format(number,parent_number,type_paragraph)
         if !ret
            errors.add(:base, "Napaka v kombinaciji fieldov number,
                                parent_number, type_paragraph")
            return false
         end
         return ret
      end
```

Metoda preveri, če ima razdelek pravilno definiran razdelek očeta. Preveri tudi, če smo za določeno številko razdelka izbrali pravi tip razdelka. Za razdelek s številko "`1.1`" mora biti izbran tip razdelka `subchapter`. S tem smo preverili pravilnost relacij med razdelki pri vnosu razdelka. Poglejmo si sedaj še, kako je to videti v praksi. Vnesemo neveljaven primer – poglavje 3 v našem primeru v diplomi še ne obstaja:

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO



Slika 14. Primer napake pri vnosu razdelka

4.5 Prikazovanje spletnih strani v slovenskem jeziku

V tem razdelku se bomo posvetili prikazovanju spletnih strani v slovenščini. Trenutno je naša aplikacija napisana tako, da se imena stolpcev tabele razdelkov, imena gumbov na spletnih obrazcih in sporočila napak na spletnih straneh izpisujejo v angleščini.
Proces, s katerim prevedemo izraze na spletnih straneh v katerikoli izbrani jezik, se imenuje lokalizacija. V okolju Rails obstaja paket imenovan `I18n`, v katerem so združene metode, ki jih uporabljamo pri prevajanju vsebine spletnih strani. Proces lokalizacije aplikacije v okolju Rails je sestavljen iz naslednjih nalog:

- Določanje lokalnega jezika – vsebuje metode, s katerimi določimo jezik, v katerem se bo prikazovala vsebina spletnih strani naše aplikacije. Izbira jezika se shrani v spremenljivko krovnega razreda `I18n`, imenovano `I18n.locale`.
- Prevajanje izrazov – postopek, s katerim posameznim izrazom, ki so deli vsebine spletnih strani, priredimo njihove prevedene različice, ki ustrezajo izbiri jezika, ki smo ga shranili v spremenljivko `I18n.locale`.

Obstaja več postopkov, s katerimi določamo jezik, v katerem bo vrnjena vsebina spletnih strani. Uporabnik denimo lahko za vsako spletno stran, ki jo želi prikazati, doda spletnemu naslovu končnico `?locale=sl`. S tem določi, da želi prikaz strani v slovenskem jeziku. Seveda je to nekoliko nepraktično. V večjih aplikacijah zato na uvodni spletni strani obstajajo povezave, s katerimi določimo jezik, v katerem se bodo prikazovale vse spletne strani, do katerih dostopamo iz uvodne strani. V bistvu se v ozadju izvede dodajanje zgornje končnice naslovom spletnih strani. Pri naši aplikaciji želimo strani prikazovati samo v slovenskem jeziku. To storimo z dodajanjem posebne metode v krmilnik razdelkov `Paragraphs` naše aplikacije. Tip posebne metode, ki jo bomo uporabili, se imenuje filter. Filtre smo spoznali v razdelku *Rails, Opis delov krmilnika za izvajanje posebnih nalog aplikacije*. V naš krmilnik `Paragraphs` bomo dodali filter metodo `set_locale`:

```
class ParagraphsController < ApplicationController
  before_filter :set_locale
```

```
...
def set_locale
   I18n.locale = :sl
end
```

S tem smo določili, da se slovenski jezik uporablja v vseh akcijah krmilnika. S tem smo zaključili s spremembami v kodi, povezanimi z določanjem jezika, v katerem bomo prikazovali spletne strani naše aplikacije.

Poskrbeti moramo še za prevode izrazov, ki sestavljajo vsebino spletnih strani. Izraze bomo prevedli iz angleškega v slovenski jezik. Zato moramo ustvariti novo datoteko `sl.yml`. Datoteko bomo ustvarili v imeniku `/home/andrej/ruby/diploma/config/locales/`. V ta imenik shranjujemo datoteke s prevodi izrazov za posamezne jezike, ki jih podpira naša aplikacija. V vsakem primeru v tem imeniku obstaja datoteka za angleški jezik, ki se imenuje `en.yml`. Datoteke `.yml` so napisane po posebnem vzorcu. Služijo kot konfiguracijske datoteke za krovni razred paketa `I18n`. Z nastavitvami, ki so v njih, metodam razreda `I18n` podamo prevode izrazov pri določeni izbiri jezika. V datoteki navedemo prevode besedil tekstovnih elementov, ki se v vzorčnih datotekah pojavljajo v angleščini. Potrebno je prevesti vse nize, ki se v vzorčnih datotekah pišejo v angleščini in se pojavljajo na spletnih straneh. Med nje spadajo tekstovni elementi jezika HTML, naslovi gumbov in ostalih delov spletnih obrazcev, besedila napak, ki so del orodja `Flash` in imena modelov ter njihovih spremenljivk. Pri izdelavi datoteke `sl.yml` si pomagamo z vsebino datoteke `en.yml`, primer datoteke `sl.yml` s primeri prevodov pa najdemo tudi na spletu (glej *Literatura in viri*). Najprej si oglejmo vsebino ustvarjene datoteke `sl.yml`, ki jo bomo uporabili v naši aplikaciji, nato pa bomo razložili posamezne tipe vnosov v tej datoteki:

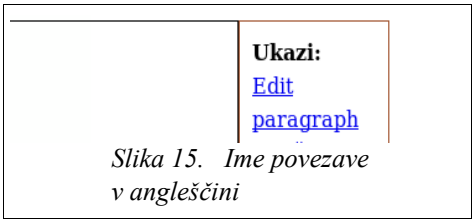
```
sl:
   "Edit paragraph": "Uredi razdelek"
   helpers:
     submit:
       create: "Ustvari nov razdelek"
       update: "Shrani spremembe"
   errors:
       template:
          header:
            one: "Obstaja napaka, ki preprečuje, da bi shranili
                  %{model}"
            two: "Dve napaki preprečujeta, da bi shranili %{model}"
            other: "%{count} napak preprečuje, da bi shranili %{model}"
          body: "Napačno izpolnjena polja:"
   activerecord:
     models:
        paragraph: "razdelek"
     attributes:
        paragraph:
          title: "Naslov"
```

```
type_paragraph: "Tip razdelka"
contents: "Vsebina"
parent_number: "Zaporedna številka nadrejenega razdelka"
number: "Številka razdelka"
```

Na začetku datoteke določimo jezik. Za slovenski jezik bomo uporabili označbo `sl:`. Nato sledijo prevodi posameznih izrazov. V vzorčni datoteki spletne strani `show.html.erb` za prikazovanje imamo denimo naslednji vstavek ERB, ki določa povezavo do spletne strani za urejanje lastnosti razdelka:

```
<%= link_to "Edit paragraph", edit_paragraph_path(@paragraph) %>
```

Samo ime se bo na spletni strani prikazalo v angleščini:



Slika 15. Ime povezave v angleščini

Če pa vstavek ERB spremenimo z uporabo makroja `t` v

```
<%= link_to t("Edit paragraph"), edit_paragraph_path(@paragraph) %>
```

dosežemo, da se uporabil prevod, ki smo ga določili v datoteki `sl.yml` v vrstici:

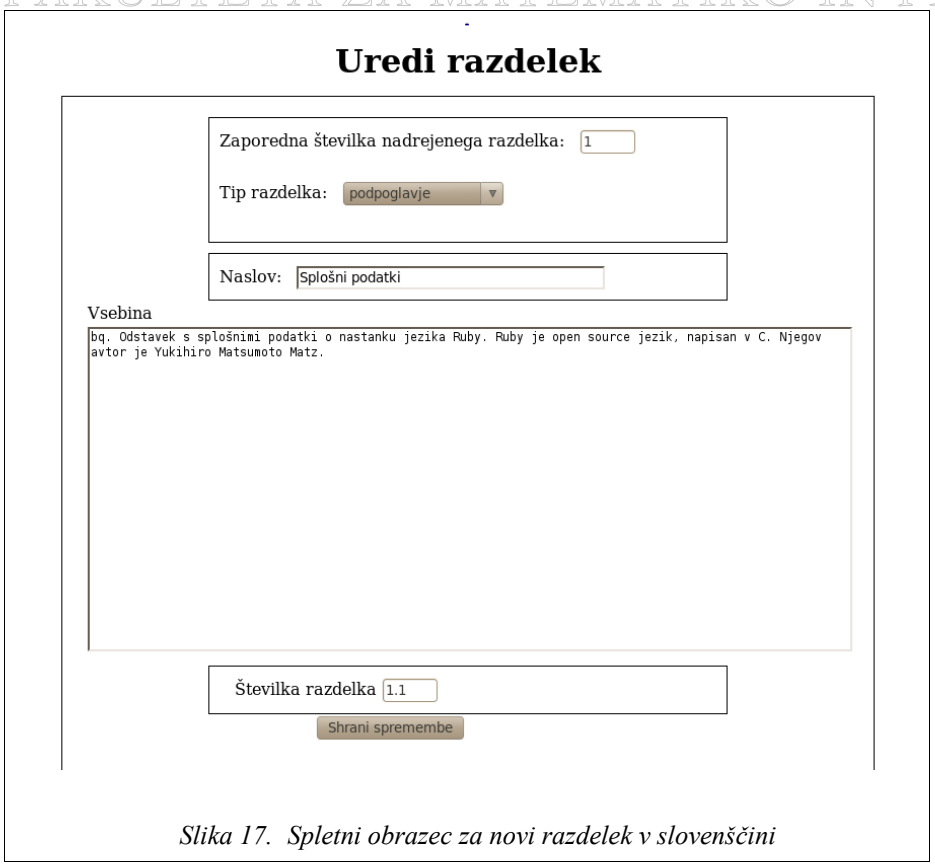
```
"Edit paragraph": "Uredi razdelek"
```

Slika 16. Ime povezave v slovenščini

Z uporabo makroja `t` prevajamo enostavne tekstovne elemente spletnih strani. Makro `t` moramo uporabiti za vse tekstovne elemente vzorčnih datotek, ki bodo vidni na generiranih spletnih straneh. V datoteko `sl.yml` pa moramo seveda dodati ustrezne prevode. V naši aplikaciji poleg enostavnih tekstovnih elementov uporabljamo tudi spletne obrazce. Z uporabo vrstic v datoteki `sl.yml`

```
helpers:
    submit:
      create: "Ustvari nov razdelek"
      update: "Shrani spremembe"
```

določimo naslov gumba za ustvarjanje novega razdelka ali urejanje obstoječega. Spomnimo se, da smo v razdelku *Rails, Spletni obrazci in skupne vzorčne datoteke* ta gumb definirali v skupni vzorčni datoteki spletnega obrazca `_form.html.erb`.
Tudi napake, ki smo jih opisali v razdelku *Rails, Opis delov krmilnika za izvajanje posebnih nalog aplikacije* želimo prikazovati v slovenskem jeziku. Prevode besedil v napakah pokriva del kode datoteke `sl.yml` pod razdelkom `errors: (glej stran 108).`
V datoteki `sl.yml` pa lahko določimo tudi imena posameznih lastnosti razdelka. Te lastnosti predstavljajo imena spremenljivk modela razdelkov `Paragraph` naše aplikacije. V datoteki `sl.yml` smo jih navedli pod razdelkom `activerecord:`.
Po tem, ko smo dodali prevode za vse izraze, ki jih vsebujejo spletne strani naše aplikacije, se spletne strani naše aplikacije prikazujejo v slovenskem jeziku. Za konec si oglejmo še sliko spletnega obrazca za urejanje razdelka, potem ko smo vse izraze prevedli v slovenščino.



Slika 17. Spletni obrazec za novi razdelek v slovenščini

5 Zaključek

S tem smo zaključili spoznavanje razvoja spletnih aplikacij v okolju Rails. Predstavili smo nekaj pomembnih lastnosti jezika Ruby in okolja Rails. Seveda pa je to le majhen delček funkcionalnosti, ki jih ponuja okolje Rails. Obstaja še veliko zanimivih lastnosti, ki jih nismo predstavili.

6 Literatura in viri

[1] Cooper Peter, *Beginning Ruby From Novice to Professional,* Berkeley: Apress, 2007

[2] Dokumentacija Bundler, *Bundler, The best way to manage your applications' dependencies,* dostopno na naslovu*:*
http://gembundler.com (zadnji dostop 14.10.2011)

[3] Dokumentacija Ruby on Rails v3.1.1, *RDOC_MAIN.rdoc*, dostopno na naslovu:
http://api.rubyonrails.org/ (zadnji dostop 14.10.2011)

[4] Fisher Timothy R., *Ruby on Rails Bible,* Indianapolis: Wiley, 2008

[5] Fitzgerald Mihael, *Learning Ruby,* Sevastopol: O'Reilly, 2007

[6] Fauser Cody, MacAulay James , Ocampo-Gooding Edward, Guenin John, *Rails 3 in a Nutshell,*

dostopno na naslovu:
http://ofps.oreilly.com/titles/9780596521424/rails.html (zadnji dostop 14.10.2011)

[7] Fuchs Sven, Rebernik Miha, *github Social Coding sl.yml (slovenska lokalizacija),* dostopno na naslovu: https://github.com/svenfuchs/rails-i18n/blob/f97137d82a0e740f03b29041ec4ef5d9aebf0007/rails/locale/sl.yml (zadnji dostop 14.10.2011)

[8] Garber Jason, *RedCloth*, dostopno na naslovu: http://redcloth.org/textile (zadnji dostop 14.10.2011)

[9] Gite Vivek, *MySQL Change root Password*, dostopno na naslovu: http://www.cyberciti.biz/faq/mysql-change-root-password/ (zadnji dostop 14.10.2011)

[10] Griffiths David, *Head first Rails / David Griffiths,* Peking [etc.] : O'Reilly,2009

[11] Harti Michael, *Ruby on Rails Tutorial Learn Rails by Example*, dostopno na naslovu: http://ruby.railstutorial.org/chapters (zadnji dostop 14.10.2011)

[12] Hipp D. Richard, *SQLite*, dostopno na naslovu: http://www.sqlite.org/sqlite.html (zadnji dostop 14.10.2011)

[13] Katz Jehuda (Wycats), *github Social Coding Thor wiki*, dostopno na naslovu: https://github.com/wycats/thor/wiki (zadnji dostop 14.10.2011)

[14] Lazarič Bojana, *Uporaba kaskadnih slogovnih predlog (CSS)*, diplomska naloga, dostopno na naslovu: http://rc.fmf.uni-lj.si/matija/OpravljeneDiplome/Lazaric-diploma_koncna.pdf (zadnji dostop 14.10.2011)

[15] Lit Steve, *Ruby Basic Tutorial*, dostopno na naslovu: http://www.troubleshooters.com/codecorn/ruby/basictutorial.htm (zadnji dostop 14.10.2011)

[16] Odsek za inteligentne sisteme, Institut "Jožef Stefan", DIS slovarček, slovar računalniških izrazov, verzija 0.8.38 dostopno na naslovu: http://dis-slovarcek.ijs.si/ (zadnji dostop 14.10.2011)

[17] Parolari Raul, I*nherited callback in Ruby*, dostopno na naslovu: http://www.raulparolari.com/Ruby2/inherited (zadnji dostop 14.10.2011)

[18] RailsGuides, *The Rails Initialization process*, dostopno na naslovu: http://guides.rubyonrails.org/initialization.html (zadnji dostop 14.10.2011)

[19] Seifer Jason, *Rake Tutorial*, dostopno na naslovu: http://jasonseifer.com/2010/04/06/rake-tutorial (zadnji dostop 14.10.2011)

[20] W3Schools, *W3Schools CSS tutorial*, dostopno na naslovu: http://www.w3schools.com/css/default.asp (zadnji dostop 14.10.2011)

[21] W3Schools, *W3Schools HTML tutorial,* dostopno na naslovu: http://www.w3schools.com/html/default.asp (zadnji dostop 14.10.2011)

[22] W3Schools, *W3Schools HTML5 tutorial*, dostopno na naslovu:

http://www.w3schools.com/html5/default.asp (zadnji dostop 14.10.2011)

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO