Bader & Jene
Software-Ingenieurbüro

RUBY ON RAILS

Eine Einführung

(aus Sicht eines Java-Programmierers)

Christian Schätzlein

Inhalt

0 Preface
1 Ruby in Kürze
- 1.1 Allgemeines
- 1.2 OO-Konzepte
- 1.3 Sprachelemente
- 1.4 Beyond

2 Ruby on Rails
- 2.1 Konzept
- 2.2 Controller
- 2.3 Model
- 2.4 View
- 2.5 Generatoren
- 2.6 Beyond

3 Fazit

0 Preface: Was ist Ruby on Rails?

Ruby on Rails (kurz RoR oder Rails) ist ein Web-Framework, das nach dem Model-View-Controller Paradigma konzipiert ist. Im Gegensatz zu anderen Web-Frameworks wie JSF oder Struts behandelt Rails auch das M in MVC, kümmert sich also um die Domain-Daten und ihre Persistenz.

RoR ist keine eigenständige Sprache, sonder lediglich ein Framework, das auf der Programmiersprache Ruby basiert. Sowohl RoR als auch Ruby sind OpenSource Projekte und stehen im Quelltext frei zur Verfügung.

Das gesamte Konzept und die Architektur von RoR ist nach den zwei Grundprinzipien

- Don't Repeat Yourself (DRY) und
- Convention over Configuration

ausgerichtet.

Bei Ersterem ist es das Ziel möglichst wenige Entwicklungs-Tätigkeiten redundant durchführen zu müssen. Sich wiederholender Code und sich wiederholende Arbeitsschritte (z.B. das Generieren von Klassengerüsten, die sich nur in wenigen Zeilen Code unterscheiden) werden im Framework vermieden oder zumindest Mechanismen bereitgestellt, um der Redundanz zu begegnen.

Das zweite Grundprinzip (Convention over Configuration) beschreibt eine weitgehende Annahme eines Standardverhaltens für zu konfigurierende Aspekte. Der Entwickler muss nur dann selbst konfigurieren, wenn das gewünschte Verhalten vom Default-Verhalten abweicht oder im konkreten Anwendungskontext nicht angewendet werden kann. Auf diese Weise werden fehleranfällige Konfigurationen auf ein Minimum reduziert und die Aufmerksamkeit des Entwicklers zurück auf die Entwicklung der eigentlichen Anwendung gelenkt.

Die strikte Verfolgung der Prinzipien hat Rails den Ruf eingebracht, mit Hilfe des Frameworks Web-Anwendungen besonders schnell und effizient realisieren zu können. Das führt dazu, dass einige „Infizierte“ sich bezüglich Rails zu amüsanten Kommentaren wie dem Folgenden hinreißen lassen:

“Ruby on Rails is astounding. Using it is like watching a kung-fu movie,where a dozen bad-ass frameworks prepare to beat up the little newcomer only to be handed their asses in a variety of imaginative ways.”

Nathan Torkington, O'Reilly Program Chair

1 Ruby in Kürze

1.1 Allgemeines

Ruby wurde 1995 zum ersten Mal publik. Zwei Jahre vorher entschloss sich der Japaner Yukihiro Matsumoto – in der Community kurz Matz genannt – in Folge einer ergebnislosen Suche nach einer objektorientierten Skriptsprache, die seinen Ansprüchen genügt, diese kurzerhand selbst zu entwickeln. Heraus kam ein Mix mit Anleihen bei etablierten Sprachen (zur damaligen Zeit) wie Smalltalk, Perl und Python: Ruby. Die Skriptsprache ist mittlerweile über die Grenzen Japans hinaus bekannt geworden und erfreut sich steigender Popularität, wozu Rails letztendlich entscheidend beigetragen hat.

Ruby ist, wie weiter oben bereits erwähnt, eine Skriptsprache. Wie für diese Sprachen typisch wird deshalb auch Ruby-Quellcode direkt von der Laufzeitumgebung interpretiert und als Ausgabe ausführbarer Maschinencode erzeugt. Der Schritt der Kompilierung in eine Zwischensprache (bei Java etwa Bytecode) entfällt demnach.

Die Vorbilder von Ruby lassen es schon erahnen: Auch Ruby bedient sich der objektorientierten Konzepte. Im Gegensatz zu Java ist Ruby sogar streng objektorientiert, d.h. alles ist ein Objekt oder anders gesagt: Es gibt keine primitiven Datentypen. Dieser Eigenschaft geschuldet, findet man im Sourcecode öfter Konstrukte, die dem Folgenden ähneln:

```
5.times { print "Wir *lieben* Ruby!" }
```

Offenbar primitiven Werten, wie Zahlen (in diesem Fall 5) werden Nachrichten gesendet. Da jedoch alles (ja, auch numerische und boolesche Werte) immer Instanzen einer Klasse sind, sprich ein Objekt sind, können und müssen alle Operationen (etwa auch Additionen, Subtraktionen etc.) über Methoden in Klassen realisiert werden.

Ein wiederum - vor allem für Java-Entwickler - gewöhnungsbedürftiger Umstand sind die fehlenden Typen. Ganz so wie es man von anderen Skriptsprachen kennt, nutzt auch Ruby untypisierte Variablen. Der wirkliche Typ einer Variablen wird deshalb erst zur Laufzeit bestimmt.

1.2 OO-Konzepte

In diesem Abschnitt soll nun beleuchtet werden, wie Ruby mit der Objektorientierung umgeht und welche Möglichkeiten bzw. Besonderheiten besonders im Vergleich mit Java bestehen.

Vererbung:

Ruby erlaubt analog zu Java nur eine einfache Vererbung. Allerdings gibt es keine Interfaces. Als Ersatz dafür dienen sogenannte Mixins mit deren Hilfe man eine Art Mehrfachvererbung realisieren kann.

Ein sehr interessanter Unterschied zu Java sind die offenen Klassen, die Ruby bietet. Sowohl alle Klassen der API als auch eigene Klassen können (ohne den Quellcode direkt zu manipulieren) bestehende Klassen erweitern. Wird eine gleichnamige Klasse geladen, überschreiben eventuell

deklarierte gleichnamige Methoden die original Methoden und zusätzlich geschriebene Methoden stehen anschließend zur Verfügung, wenn man die erweiterte Klasse benutzt. Es ist also sehr leicht möglich das Verhalten von Ruby-Klassen an die individuell für ein Projekt benötigten Eigenschaften anzupassen ohne vom Konzept der Vererbung Gebrauch zu machen.

Module:

Module sind eine Besonderheit von Ruby und stellen Sammlungen von Methoden und Konstanten bereit. Im Gegensatz zu Klassen erlauben Module keine Instanzbildung und besitzen ebenfalls keine Superklasse, können also nicht geerbt werden.

In der ersten Variante können Module Methoden und Konstanten deklarieren, die anschließend (vergleichbar mit statischen Methoden in Java) aufgerufen werden können. In der zweiten Ausprägung mit Methoden als *Mixins* können die Module über eine *include* Anweisung in Klassen eingebunden werden. Alle deklarierten Methoden stehen dann auch der Klasse zur Verfügung. Das Konzept geht sogar soweit, dass in den Modulen Annahmen über benötigte Instanzvariablen und Methoden getroffen werden können und so mit der einbindenden Klasse (exakter mit Objekten der Klasse) interagiert werden kann.

Darüber hinaus können Module verwendet werden, um eine Klassenstruktur zu erstellen, indem man in diesen Klassen platziert. Module stellen auf diese Wiese also auch einen möglichen Namensraum für Klassen zur Verfügung. Pakete, wie man es aus Java gewohnt ist, existieren nicht.

Variablen:

An Variablen offeriert Ruby die ganze gewohnte Bandbreite: Es gibt lokale Variable, Instanzvariablen, Klassenvariablen (das Pendant zu statischen Variablen in Java) und Konstanten. Daneben existieren globale Variablen, die zumindest dem Java-Programmierer neu vorkommen dürften. Globale Variablen werden an irgendeiner Stelle in der Anwendung initialisiert und stehen ab diesem Zeitpunkt auch im Rest der Anwendung ohne Weiteres zur Verfügung.

Instanzvariablen müssen in Ruby nicht im Klassenrumpf initialisiert werden. Stattdessen wird die Variable einfach bei ihrem ersten Auftreten initialisiert und steht ab diesem Zeitpunkt zur Verwendung bereit. Instanzvariablen sind zudem nur innerhalb der aktuellen Klasse und für alle Unterklassen sichtbar. Diese Art der Sichtbarkeit heißt in Ruby *private* und entspricht in etwa dem *protected* in Java. Da die private Eigenschaft der Instanzvariablen nicht änderbar ist, müssen für den Zugriff auf die Variablen außerhalb der aktuellen Klassenstruktur *Getter/Setter* geschrieben werden.

Methoden:

Im Gegensatz zu Instanzvariablen sind Methoden per Default nach Außen hin sichtbar. Sie sind also *public*. Die Sichtbarkeit kann zudem in Unterklassen jederzeit von *private* in *public* und vice versa geändert werden (sogar zur Laufzeit noch).

Die Signatur einer Methode beschränkt sich in Ruby aufgrund der fehlenden Typen lediglich auf den Namen und die Anzahl der Parameter. Überladene Methoden sind demnach per se verboten, da eine gleich lautende Nachricht (damit ist hier gleicher Name, Zahl der Parameter gemeint) an ein Objekt nicht eindeutig einer der Methoden zugeordnet werden könnte. Ruby geht sogar soweit, dass nur eine Methode mit ein und dem selben Namen existieren darf. Das Konzept der Überladung wird

deshalb anders gelöst: Über die Möglichkeit der Default-Belegung von Parametern kann eine Methode mit einer variablen Zahl an Parametern aufgerufen werden. Diese Option existiert jedoch nur für die jeweils letzten Parameter der Methode, da sonst eine eindeutige Abbildung von Parameter der Nachricht auf Parameter der Methode nicht möglich wäre. Ab der ersten Default-Angabe für einen Parameter müssen die Folgenden ebenfalls eine solche aufweisen.

Ein weiteres interessantes Konstrukt in Ruby stellen die erlaubten, mehreren Rückgabewerte für Methoden dar. Statt - wie in Java - strukturierte Werte in Objekte wrappen zu müssen, kann so einfach eine Komma-separierte Liste mit Ruby-Ausdrücken angegeben werden, die dann als Rückgabe fungiert Auf der anderen Seite ist es dann natürlich auch möglich eine Liste an Variablen zu deklarieren, denen die zurückgegeben Referenzen zugewiesen werden.

Beispiel

Zur Veranschaulichung der Ruby-Syntax ist im Folgenden der vollständige Quelltext einer Ruby-Klasse abgebildet:

```ruby
class Programmer < Person
  #globale Variable
  $ruby_inventor = "Yukihiro Matsumoto"

  #Klassenvariable
  @@nr_of_programmers = 0

  #Konstanten
  STRESSED = 1
  LUCKY = 2

  #Short Cut für getter und setter
  attr_accessor :language

  #Konstruktor-Methode
  def initialize name, age, language, mood=STRESSED
    super(name, age)
    @language = language
    @mood = mood
    @@nr_of_programmers += 1
  end

  #getter
  def mood
    @mood
  end

  #setter
  def mood= mood
    @mood = mood
  end
end
#lokale Variable
pr = Programmer.new "Jim Knopf", "51", "Ruby", Programmer::LUCKY
puts pr.name # => Jim
pr.language = "Java"
puts pr.language # => Java
```

Die Deklaration einer Klasse, sowie der Vererbung gestalten sich unkompliziert (Zeile1). Angezeigt vom Schlüsselwort *class* folgt zunächst der Name der aktuellen Klasse und anschließend die Superklasse (nach dem < Zeichen). Die gesamte Klasse ist eingeschlossen in einen Block, der über ein *end* beendet wird (Zeile 32).

Nach dem Klassenrumpf folgt die Angabe der Variablen, die initialisiert werden müssen. Eine globale Variable wird dabei durch ein Dollar-Symbol markiert (Zeile 3). Klassenvariablen hingegen sind durch ein doppeltes @ ausgezeichnet (Zeile 6). Konstanten werden ebenfalls nicht durch einen „Modifier“ oder etwas Ähnlichem symbolisiert. Stattdessen werden Variablen einfach durch die konsequente Großschreibung des Namens zu einer Konstanten (Zeile 9,10). Der Zugriff erfolgt dann über den Klassennamen und mit Hilfe der speziellen Notation mit zwei Doppelpunkten (Zeile 34).

Analog zu Eiffel oder Smalltalk ist auch in Ruby der Konstruktor einfach eine Methode *(initialize* Zeile 16*)*. Im dargestellten Beispiel besitzt diese zudem noch einen optionalen Parameter, wodurch ein *Programmer* entweder unter Angabe der Gefühlslage (*mood*) oder ohne erzeugt werden kann. In letzterem Fall wird dann standardmäßig der Wert der Default-Belegung *(STRESSED)* für die Variable angenommen.

Wie in objektorientierten Sprachen üblich muss der Konstruktor der Superklasse aufgerufen werden. Analog zu Java erfolgt auch in Ruby dieses über das Schlüsselwort *super* (Zeile 17). Wird dabei nicht explizit ein parametrisierter Konstruktor aufgerufen, findet automatisch der leere Konstruktor der Oberklasse Verwendung.

Innerhalb der *initialize* Methode begegnet man den ersten Instanzvariablen *(language* und *mood, Zeile 18, 19)*. Im Kontrast zu Klassenvariablen werden diese allerdings nur durch ein einfaches @ als Präfix im Namen symbolisiert. Da (wie im Abschnitt zu den Variablen erwähnt) keine Initialisierung im Klassenrumpf vorgenommen werden muss, sind die Variablen erst nach der jeweiligen Anweisung ansprechbar.

Um nun die initialisierten Instanzvariablen auch für Objekte zugänglich zu machen, die lediglich eine Referenz auf das „Objekt der Begierde“ halten, müssen *Getter/Setter* geschrieben werden. Per Konvention heißt die Methode für die *Getter* dabei genauso wie die Variable selbst (Zeile 24 für Variable *mood*) und die *Setter*-Methode wie eine Zuweisung der Variablen (Zeile 29). Die Benutzung der Methoden liest sich somit, als ob man direkt mit den Instanzvariablen arbeiten würde. In Wahrheit sind es jedoch Methoden, die die Arbeit übernehmen. Und Ruby wäre nicht Ruby, wenn es nicht einen effizienteren Weg geben würde, um die *Getter/Setter* bereitzustellen. Anstatt die Methoden jedes mal manuell zu programmieren, können Short-Cuts benutzt werden. Diese sind nichts anderes als Klassenmethoden, denen die Namen der Instanzvariablen übergeben werden und die dann die benötigten Methoden zur Verfügung stellen. Short-Cuts gibt es für schreibenden *(attr_writer)*, nur lesenden *(attr_reader)* und kombinierten Zugriff *(attr_accessor,* z.B. in Zeile 13 für die Variable *language)*.

Als letztes soll die Aufmerksamkeit auf das Ende des Beispiels gelenkt werden. Dort sieht man außerhalb der Klassendefinition weiteren Quelltext (Zeile 34-37). In Java würde diese Platzierung zu einem Fehler bei der Kompilierung führen. In Ruby werden die Anweisungen stattdessen beim Ausführen des Ruby-Skriptes dem Auftreten nach ausgeführt. Eine *main* Methode wie man sie aus Java kennt, gibt es demnach nicht.

1.3 Sprachelemente

Ruby bietet viele bekannte syntaktische Konstrukte. Aus Java-Sicht betrachtet, existieren jedoch auch zahlreiche neuartige und ungewohnte Möglichkeiten. An dieser Stelle soll nun ein Blick auf einige ausgewählte Sprachelemente geworfen werden, die vor allem im Hinblick auf Ruby on Rails wichtig sind.

Blöcke

Blöcke sind Code-Fragmente bzw. Code-Blöcke, die einer Methode übergeben werden. Die Methode kann dann an einer beliebigen Stelle über das Schlüsselwort *yield* den Block ausführen. Optional ist es zudem möglich dem Block mit Parametern aufzurufen.

```
def execute(count)
  puts count
  yield
end

execute(42) { puts "ausgeführt!" }
# => 42
# ausgeführt!
```

Im obigen Beispiel für die Verwendung von Blöcken wird eine Methode definiert, die lediglich eine Ausgabe auf der Konsole erzeugt und anschließend einen übergebenen Block ausführt.

Schleifen

Ruby bietet verschiedene Arten von Schleifen bzw. Möglichkeiten zum Iterieren. Unter diesen sind die obligatorischen *while* und *for* Schleifen. Letztere veranschaulicht das nachstehende Beispiel:

```
for item in item_set do
  puts item
end
```

Die Syntax ist intuitiv und bedarf keiner große Gewöhnung. Iteriert werden kann über alle Kollektions-Objekte (im Beispiel *item_set*), die eine *each* Methode implementieren. Diese kann natürlich über Blöcke auch direkt verwendet werden. Weitere Konstrukte in Ruby zum Erzeugen von Schleifen/Iterieren sind u.a.: *until* (wie *for* nur negierte Bedingung), *times* (für Bignum/Fixnum Objekte) und *upto* (für Strings/Dates/Bignum/Fixnums*).

Symbole

Symbole werden in Ruby in bestimmten Situationen als Alternative zu Strings verwendet. Sie kommen meist überall dort zum Einsatz, wo keine String-Operationen (Länge eines Strings ausgeben, Splitting, Case-Konvertierung etc.) von Nöten sind, sondern ledigliglich ein Konstrukt zur Benennung. So finden Symbole zum Beispiel Anwendung als Schlüssel in HashMaps oder werden zur Bezeichnung von Schlüsselwörtern herangezogen, um etwa Optionen für Methoden abzubilden.

```
:symbol_name
```

* Bignum/Fixnum repräsentieren Integer-Zahlen mit unterschiedlichen Beriechen

Das Besondere an Symbolen ist dabei ihre implizite Singleton-Eigenschaft: Egal an welcher Stelle man ein Symbol im Programm benutzt, es handelt sich immer um das gleiche Objekt. Darüber hinaus können Symbole nicht manipuliert werden.

Arrays

Die Verwendung von Arrays gestaltet sich in Ruby sehr ähnlich zu anderen objektorientierten Programmiersprachen. Über eckige Klammer oder den *new* Operator können zunächst leere Arrays erzeugt werden (Zeile 1,2). Der Zugriff wird ebenfalls über solche Klammern gelöst, wobei innerhalb dieser dann ein Index bestimmt, an welcher Stelle im Array das Objekt gespeichert wird bzw. welche Position zurückgegeben werden soll (Zeile 3). Eine Besonderheit Rubys ist, dass als Index auch negative Werte akzeptiert werden. Möchte man einfach nur ein Element oder gar ein ganzes Array an das Ende eines exisiterenden Arrays anhängen, bewerkstelligt dieses der << Operator (Zeile 5). Mit den sogenannten *Ranges* (ein Bereich bestimmt durch eine obere und untere Grenze) kann darüber hinaus ein Subarray aus dem Aktuellen extrahiert werden (Zeile 7).

```
1 array = Array.new # []
2 empty = []        # []
3 array[0] = 42     # [42]
4 numbers = [1, 7]  # [1, 7]
5 numbers << 2      # [1, 7, 2]
6 numbers[0]        # 1
7 numbers[1..2]     # [7, 2]
```

HashMaps

Die Arbeit mit HashMaps erweist sich in Ruby ähnlich unkompliziert, wie die mit Arrays. Analog kann ein neues, leeres Hash-Objekt entweder explizit über den *new* Operator oder über ein leeres Klammernpaar instanziert werden (Zeile 1,2). HashMaps werden dabei im Gegensatz zu Arrays durch geschweifte Klammern symbolisiert (Zeile 2, 4). Der Zugriff auf Einträge erfolgt allerdings wieder über eckige Klammern. Anstelle des Indexes tritt an dieser Stelle jedoch der Schlüssel für den Hash-Eintrag (Zeile 3, 5). Für die Erzeugung einer HashMap mit einer intialen Belegung kommen Symbole zum Einsatz. Die Abbildung vom Schlüssel auf den Wert wird hierbei recht gut durch das Pfeil-Konstrukt deutlich (Zeile 4).

```
1 members = Hash.new           # {}
2 empty = {}                   # {}
3 members[2] = "Thomas"        # { 2 => Thomas }
4 hash = {:x => 199, :z => 2 }# {x => 199, z => 2 }
5 hash[:x]                     # 199
```

1.4 Beyond

Die bereits vorgestellten Eigenschaften und Möglichkeiten der Programmiersprache Ruby erheben natürlich keinen Anspruch auf Vollständigkeit. Ruby ist eine mächtige Sprache, die über viele weitere Features (Syntax, APIs) verfügt. Die folgende Liste enthält daher nur einen kleinen Teil an weiteren, wichtigen und interessanten Aspekten. Für eine ausführlichere Betrachtung dieser und weiterer Themen sei an dieser Stelle auf die Literatur verwiesen.

Exception Handling

- Ruby bietet ebenso wie Java eine Ausnahmebehandlung. Allerdings heißen die Schlüsselwörter hier nicht *throw/catch* sondern *raise* und *rescue*. Der potenziell Fehler produzierende Code wird dabei anstelle eines *try Blocks* einfach in einen *begin/end* Block gefasst.*

Automatische Speicherverwaltung

- Ruby-Entwickler müssen sich nicht um verwaiste Objekt-Referenzen sorgen. Ein Garbage-Collector übernimmt das Aufräumen des Speichers automatisch.

Reflection

- Das Wort „Reflection“ ist an dieser Stelle eigentlich falsch. Ein jeder Java-Entwickler weiß jedoch sofort, was unter diesem Feature zu verstehen ist. In Ruby verbirgt sich Ähnliches unter dem Synonym „Introspection“: Das Erfragen von Klassenhierarchien, akzeptierten Nachrichten und Eigenschaften von Klassen bzw. Objekten und vielem mehr ist somit kein Problem.

Threads

- Ruby besitzt eine eigene Thread-API, die nicht auf der unterliegenden Plattform aufsetzt, sondern vollständig in Ruby implementiert ist.

Datenbank-Zugriff

- Zur Interaktion mit Datenbanken hält Ruby zahlreichere Möglichkeiten bereit. So existiert etwa ein Bibliothek zum expliziten Zugriff auf MySql-Datenbanken, sowie Ruby's *Database Interface* (DBI). Letzteres bietet dabei eine Abstraktion vom konkret verwendeten DBMS und kann als ein Äquivalent zu *JDBC* gesehen werden.

Netzwerk-Zugriff

- Der Netzwerk-Zugriff kann in Ruby auf Socket-Ebene erfolgen oder über höhere Protokolle z.B. HTTP, FTP.

Web-Framework

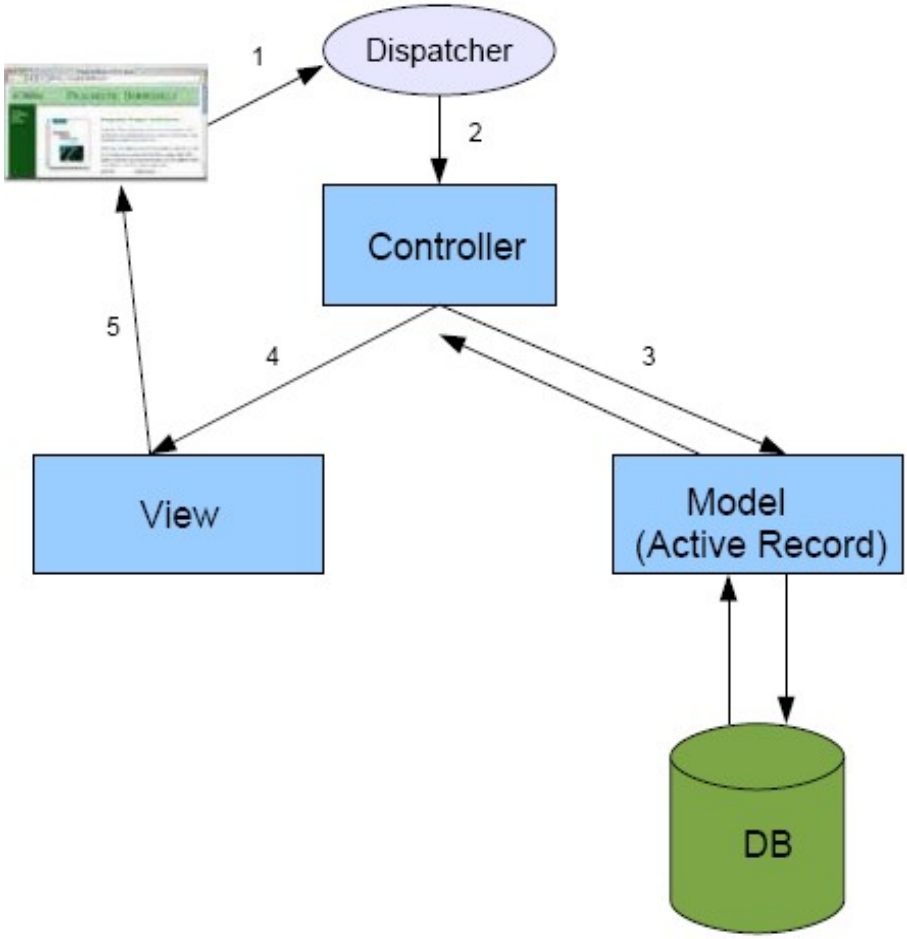
- Neben all den schönen Features existiert auch ein in Ruby geschriebenes Framework zum Entwickeln von Web-Anwendungen: **Ruby on Rails**.

* Das *throw/catch* Konstrukt gibt es auch in Ruby. Die Semantik ist hier jedoch eine Andere.

2 Ruby on Rails

2.1 Konzept

Rails folgt in der Architektur dem Model-View-Controller Paradigma. Dieses führt durch die explizite Unterscheidung in Präsentation (View), Daten (Model) und Ablaufsteuerung (Controller) ein saubere Trennung von Zuständigkeiten ein. Ziel ist es, die eigentliche Entwicklung der Anwendungslogik bzw. die technische Seite vom Design des Frontends (zumeist HTML) zu entkoppeln und die Datenschicht so unabhängig wie möglich vom konkreten Persistenz-Mechanismus zu gestalten. Rails folgt diesem Ansatz, wobei im Gegensatz zu anderen Web-Frameworks (z.B. Struts, JSF) auch der Umgang mit den Domaindaten und ihre Persistenz (sprich das M in MVC) integriert ist.



Das obige Bild skizziert das Konzept bzw. den Ablauf eines typischen Request/Response Zyklus. Anhand der Nummerierungen werden im Folgenden die einzelnen Schritte genauer beschrieben:

(1) Ein Client gibt z.B. eine URL im Browser ein und produziert einen HTTP-Request.

(2) Ein ankommender Request wird von einem CGI-Skript entgegengenommen und routet diesen zum adressierten Controller.

(3) Der Controller nimmt den Request entgegen und interagiert mit dem Modell, um bestimmte Entitäten zu finden und Aktualisierungen vorzunehmen. Alle Veränderungen im Modell werden dabei automatisch in einer Datenbank persistiert.

(4) Im Anschluss an die Interaktion mit dem Modell, initiiert der Controller den Response indem ein View ausgewählt, mit den Domaindaten (Modell-Objekten) gespeist und anschließend gerendert wird (Injektion des dynamsichen Anteils aus dem Modell).

(5) Die Ausgabe eines gerenderten Views ist HTML und wird als Response an den Client zurückgesendet.

2.2 Controller

Der Controller ist für die Steuerung des Ablaufes innerhalb der Anwendung dar. Programmatisch erbt jeder Controller von der Klasse *ActionController::Base*. Welcher Controller aufgerufen werden soll, entscheidet der Dispatcher anhand der URL. Darüber hinaus enthält diese auch die Information welche Methode der Controller-Klasse aufgerufen werden soll. Eine solche Methode bzw. Nachricht an ein Controller-Objekt wird in Rails als *Action* bezeichnet. Die Abbildung von URL auf Action folgt dabei per Default immer der Konvention, dass der erste Teil in der Adresse den Controller und der Zweite die Action bestimmt. Vom optionalen dritten Teil wird angenommen, dass dieser eine ID für eine Entität darstellt. Das nachstehende Beispiel veranschaulicht diese Abbildung noch einmal:

URL = http://user/show/1 Controller => User Action => show, id =>1

Der Zugriff auf relevante Daten der Verbindung erfolgt über HashMaps. Beispielweise erfolgt die Abfrage der Parameter des HTTP-Requests immer über die Variable *params,* die für jede Action zur Verfügung gestellt wird. Analog sind auch Session und Cookie-Daten zugänglich.

Hat der Controller alle Interaktionen mit dem Modell abgeschlossen, gibt es drei Alternativen: Entweder in der Action wird explizit eine View gerendert oder der gesamte Request wird an einen anderen Controller bzw. eine andere Action weitergeleitet *(redirected).* Wird weder das eine noch das andere explizit vorgenommen, greifen die Konventionen von Rails. Standardmäßig wird in diesem Fall einfach eine View gerendert, die den gleichen Namen wie die Action trägt und sich in im View-Verzeichnis des Controllers befindet. Zur Veranschaulichung wiederum ein Beispiel:

URL = http://user/list View => list.rhtml Verzeichnis => views/user

2.3 Modell

Die Model-Komponente ist eine der interessantesten Seiten an Ruby on Rails. Auf Code-Ebene ist jedes Modell dabei eine Unterklasse von *ActiveRecord::Base.* Diese Klasse ist nichts anderes als ein

eingebauter O/R-Mapper. Für die Arbeit mit unterschiedlichen DBMS stehen zahlreiche Adapter bereit, die die eigentliche Arbeit im Hintergrund erledigen. Zur Zeit existieren Adapter für die folgenden DBMS: DB2, Firebird, MySQL, OpenBase, Oracle, PostgreSQL, SQLServer, SQLite and Sybase.

Die Abbildung von Modell auf Datenbank-Tabellen ist durch Konventionen geregelt. Der Name des Modells ist demnach im Singular und „Pascal Style“ (auf diese Weise werden auch Java-Klassennamen notiert) zuhalten, während die Tabellen im Plural und typischen Datenbank-Stil benannt werden. Für eine Entität *Important Model* sehen die Bezeichnungen dann z.B. wie folgt aus:

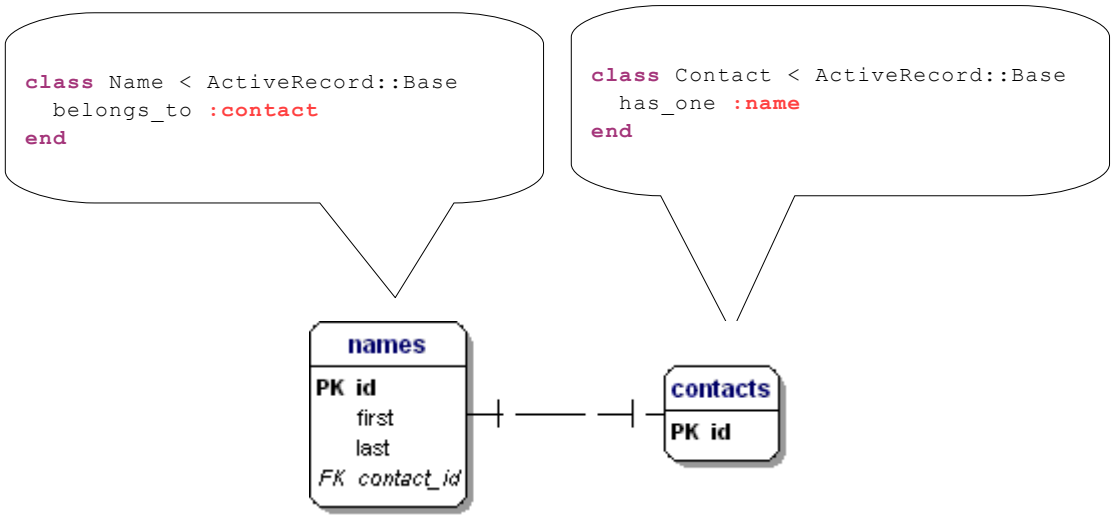
Model = ImportantModel DB-Tabelle = important_model

Ist man den Konventionen gefolgt und hat sowohl Model als auch Datenbank-Tabelle erstellt, können ohne weiteren Aufwand CRUD-Aktionen (Create, Read, Update, Delete) ausgeführt werden. Die dafür nötigen Methoden werden von ActiveRecord bereitgestellt. Darüber hinaus können die Attribute der Datenbank-Tabelle über *Getter/Setter* abgefragt bzw. gesetzt werden. Außerdem kann auf einfache Art und Weise nach bestimmten Entitäten gesucht werden.

Eine Frage die nun im Raum steht ist: Wie behandelt Rails Assoziationen und Vererbung? Überraschenderweise lautet die Antwort wieder über Konventionen. Assoziationen werden dabei immer im Model beschrieben.

1:1 Beziehungen

Um eine 1:1 Beziehung zu realisieren müssen die beteiligten Tabellen bzw. Modelle zunächst der Namenskonvention folgen. Außerdem muss in einer der beteiligten Tabellen ein Fremdschlüssel existieren. Dieser setzt sich per Default aus dem Namen der Tabelle (Singular) und dem Suffix „id“ zusammen. Auf Datenbank-Seite ist weiter nichts zu tun. Auf Modell-Ebene dagegen muss die Assoziation über Klassenmethoden deklariert werden. Dafür stehen bei 1:1 Beziehungen die Methoden *belongs_to* und *has_one* zur Verfügung. Diesen werden dann als Parameter die Namen der Assoziation übergeben (Singular der Tabellen). Der Unterschied zwischen den beiden Methoden besteht darin, dass *belongs_to* die Tabelle mit dem Fremdschlüssel anzeigt.



Sind die Tabellen und die Modelle entsprechend erzeugt, ist das O/R-Mapping auch schon komplett konfiguriert. Auf Code-Ebene navigiert man nun wie gewohnt auf den Modellen. Für das obige Beispiel muss dafür zunächst über die zur Verfügung gestellten Such- bzw. Erzeugungsmethoden eine Instanz an eine Variable gebunden werden. Anschließend erhält man über die Punktnotation die assoziierten Objekte:

contact.name bzw. *name.contact*

Die Zuweisung der Beziehungen erfolgt über die automatisch bereitgestellten *Setter*:

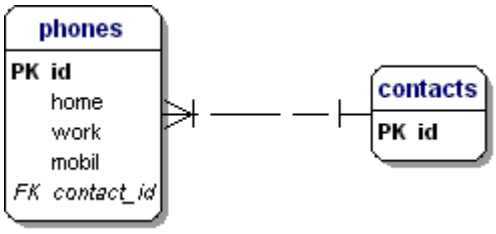
contact.name= new_name

1:n Beziehungen

Die Behandlung von Assoziationen mit nur einem mehrfachen Ende unterscheidet sich bei Ruby on Rails nur unwesentlich von der des kleinen Bruder, den 1:1 Beziehungen. Natürlich müssen jedoch auch hier die beteiligten Tabellen und Modelle den Konventionen folgen. Die Tabellenstruktur ist dabei analog zur einfachen Assoziation aufgebaut. Der Fremdschlüssel befindet sich selbstverständlich auf der N-Seite und besteht wiederum aus dem Tabellennamen plus dem Suffix *id*.

```
class Phone < ActiveRecord::Base
  belongs_to :contact
end
```

```
class Contact < ActiveRecord::Base
  has_many :phones
end
```



Auf Modellebene ändert sich lediglich die Deklaration der mehrfachen Assoziation. Anstatt einem *has_one* symbolisiert nun ein *has_many* auf der 1-Seite der Beziehung, dass dieses Modell viele Assoziationspartner besitzt. Die eindeutige Zuordnung der N-Seite wird dagegen weiterhin durch ein *belongs_to* angezeigt.

Für die Navigation bzw. Zuweisung sind die Veränderung ebenso einseitig. Das Setzen und Abfragen der 1-Seite ist demnach weiterhin durch einfache *Getter/Setter* realisert, wobei der *Setter* genau ein Domain-Objekt als Parameter erwartet.

Anders verhält es sich mit der N-Seite, da hier nunmehr eine Menge an Objekten geliefert wird (im obigen Beispiel etwa mehrere Phone-Objekte). Die Menge an Assoziationspartnern gibt Rails in Form einer speziellen Kollektion zurück. Diese ermöglicht neben dem Iterieren u.a. auch das Erzeugen und Löschen der assoziierten Objekte.

Bei der Initialisierung der Beziehung erwartet der *Setter* diesmal als Parameter allerdings ein Array an Domain-Objekten und nicht ein einzelnes:

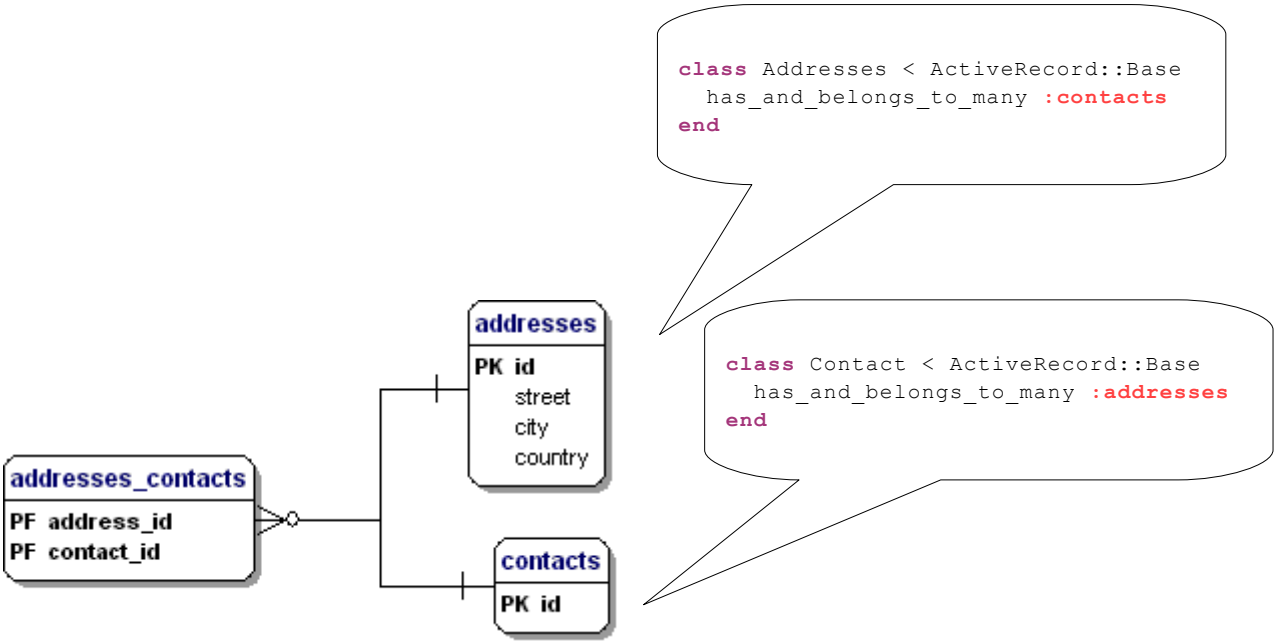
```
contact.phones = [phone1, phone2, phone3]
```

Die spezielle Kollektion erlaubt in Array-Manier ebenfalls das Hinzufügen eines Objektes zur Kollektion:

```
contact.phones <<  phone4
```

n:n Beziehungen

Die dritte der möglichen Beziehungstypen sind die n:n Beziehungen, sozusagen der König unter den Beziehungen. So überrascht es nicht, das ein wenig mehr Aufwand notwendig ist, um eine solche Assoziation mit Rails zu realisieren. Doch zunächst steht wiederum das Datenbank-Schema im Focus. In relationalen Datenbanken werden n:n Beziehungen über eine dritte Tabelle, der *Join-Table* realisert. Diese Tabelle beinhaltet jeweils den Primärschlüssel der beteiligten Tabellen als Fremdschlüssel. Per Konvention werden die Schlüssel wie zuvor aus dem Singular des Tabellennamens und dem Suffix „id“ zusammengestzt. Der Name für die Join-Table selber wird aus den beiden beteiligten Tabellennamen gebildet. Die Reihenfolge ist dabei alphabetisch zu wählen.



Auf Modellebene werden n:n Beziehungen über die Klassenmethode *has_and_belongs_to_many* deklariert. Die Navigation bzw. Zuweisung gestaltet sich nun genauso, wie für die N-Seite bei 1:n

Beziehungen, mit dem Unterschied, dass dies für beide Assoziationsenden gilt. Für beide bekommt man also eine Kollektion geliefert, die u.a. das Löschen, Erzeugen und Iterieren von Assoziationsobjeken ermöglicht.

Möchte man in der Join-Tabelle weitere Spalten ansiedeln, sprich also der Assoziation bestimmte Attribute zuordnen, kann das über eine 2-stufige Assoziation erreicht werden. Zunächst wird dafür eine 1:n Beziehung mit der Join-Tabelle spezifiziert (für *contacts* aus dem obigen Beispiel):

```
has_many :addresses_contacts
```

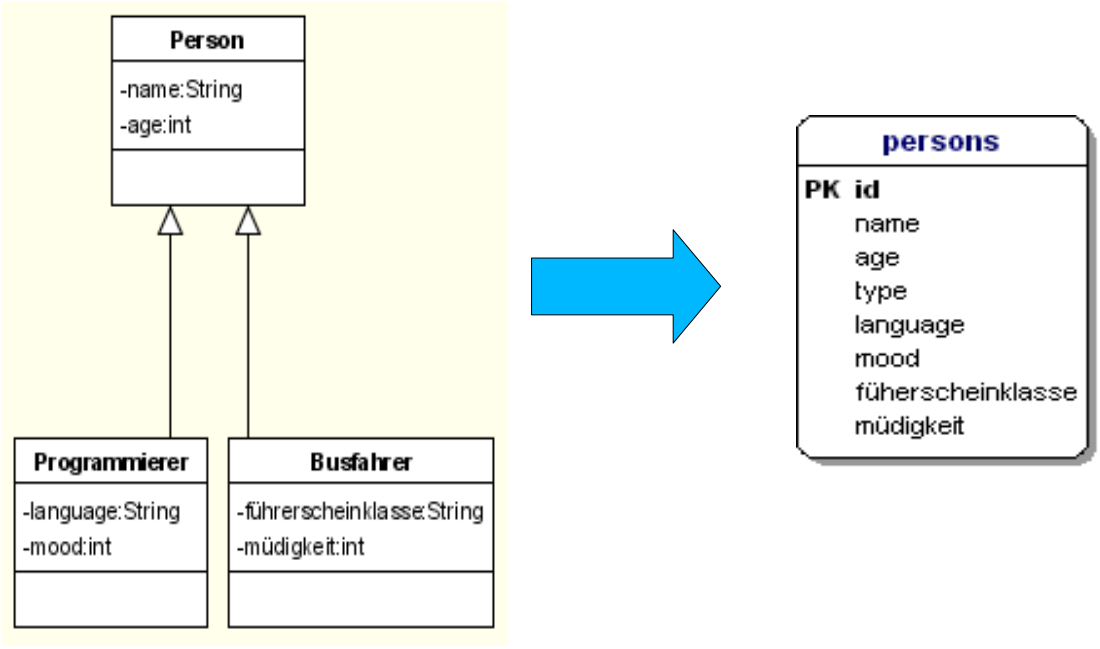
Dann wird zusätzlich über eine 1:n Beziehung mit der eigentlichen Zieltabelle angegeben, dass diese über die Join-Tabelle zu knüpfen ist:

```
has_many :addresses, :through => :addresses_contacts
```

Vererbung

Ein weiterer Punkt den es zu untersuchen gilt, ist wie Ruby on Rails mit dem Thema Vererbung umgeht? Die Antwort ist, dass Rails (oder genauer ActiveRecord) im Gegensatz zu anderen O/R Mappern wie Hibernate nur eine eine einzige Methodik anbietet, um eine Klassenhierarchie auf Datenbank-Ebene abzubilden: Die so genannte „SingleTable Inheritance“

In Rails werden so alle Unterklasse zusammen mit der Oberklasse in einer Tabelle gespeichert. Als Spalten besitzt diese Tabelle die Vereinigung aller Attribute der relevanten Klassen. Ein besonderes, zusätzliches Attribut (per Default *type*) wird dann verwendet, um den konkreten Typ der Klasse zu speichern.



Der Nachteil an diesem Verfahren ist, dass Attribute der Unterklassen nicht als Spalten mit einem *Not Null Constraint* in der Tabelle umgesetzt werden können. Verletzt man diese Regel, könnte als Folge dessen keine Instanz einer anderen Unterklasse erzeugt werden, da diese die Constraint

immer verletzten würde (sie besitzt kein derartiges Attribut). Zur Veranschaulichung könnte im abgebildeten Beispiel kein *Programmier* instanziert werden, wenn in der Tabelle eine Spalte des *Busfahrers* (etwa *müdigkeit*) mit einem *Not Null Constraint* belegt ist, da ein Programmierer dieses Attribut nicht aufweist und somit dieses auch nicht mit einem anderen Wert als *Null* belegt werden kann. Dieses Problem stellt allerdings keine wirkliche Einschränkung dar. Möchte man erzwingen, dass ein Attribut des Modells mit einem Wert belegt wird, kann dies auf einfache Weise über die eingebaute Validierungskapazitäten von Ruby on Rails erreicht werden.

Validierung

Die Validierung von Eingabedaten eines Formulars wird im Modell deklariert. Zu diesem Zweck existieren im Rails-Framework bereits vordefinierte Validatoren in Form von Klassenmethoden. Diesen wird einfach der Name des zu validierende Attributs in Form eines Symbols übergeben. Einige dieser Standardvalidatoren sind im Folgenden aufgelistet:

- *validates_length_of* - überprüft, ob etwa ein String eine bestimmte Länge hat
- *validates_numericality_of* - stellt sicher, dass es sich um einen numerischen Wert handelt
- *validates_presence_of* - Pendant zum *Not Null Constraint* (Attribut kann nicht *nil** sein)
- *validates_format_of* - überprüft Format anhand eines regulären Ausdrucks

Wem die Palette der vordefinierten Validatoren nicht ausreicht, kann selbstverständlich auch seine eigene Validierungen vornehmen, Dazu können zum Einen eigene Validatoren dank Rubys offener Klassen zu den bestehenden hinzugefügt werden oder in einer *validate* Methode des Modells die Validierung vollzogen werden.

Bevor ein Objekt persistiert wird, wird zunächst die Validierung durchgeführt. Dabei auftretende Fehler werden in ein ErrorObjekt gewrappt und dem Modell-Objekt zugewiesen. Die Errors können dann anschließend im View ausgelesen und visualisiert werden.

2.3 View

Die View ist für die Präsentation der Web-Anwendung zuständig. Die dynamischen Anteile werden im Render-Prozess serverseitig durch statisches HTML substituiert und schlussendlich die Ausgabe erzeugt. Dementsprechend handelt es sich bei einer View lediglich um ein HTML-Gerüst mit eingebettetem Ruby-Code.

Sucht man vergleichbares in der Java-Welt, so findet man bei den Java Server Pages ein sehr ähnliches Konzept. So bietet auch Rails etwas wie Srciptlets (eingeschlossen in <% %>) und Expressions (eingeschlossenen in <%= %>). Erstere werden benutzt um Schleifen, Verzweigungen u.a. Ruby Code auszuführen. Letztere enthalten dagegen einen Ruby-Ausdruck, der dann in die Ausgabe übernommen wird. Folgendes Beispiel verdeutlicht Syntax und Semantik:

```
<% 3.times do %>
<%= @person.name %>
<% end %>
```

* Rubys Äquivalent zu *null* in Java

Im abgebildeten Beispiel-Code wird eine Schleife erzeugt, in dessen Körper dann eine Instanzvariable *person* nach einem Attribut *name* gefragt wird (via der *Getter*-Methode). Als Ausgabe würde in diesem Fall dreimal der zurückgegebene Name erscheinen.

Instanzvariablen (wie im obigen Beispiel), die in einer Action eines Controllers instanziert wurden, stehen anschließend auch in der View zur Verfügung. Auf diese Weise können der View Domain-Daten übergeben werden, die dann aufbereitet und visualisiert werden.

Für den Umgang mit HTML stellt Rails Hilfsmethoden bereit. Zahlreiche Konstrukte können so einfach über eine Methode generiert werden, anstatt jedes mal das HTML selbst coden zu müssen..Alle Methoden sind dabei in sogenannten Helper-Klassen angesiedelt. Die folgende Liste stellt einige dieser Klassen vor:

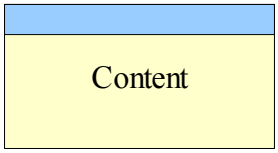
- FormHelper
  - stellt Methoden für die Arbeit mit Formularen bereit (Textfelder, Check-Boxen usw.)
  - ausgelegt für Formularfelder, die zu einem Attribut eines Modells gehören
  - Bsp. **text_field**(object_name, method, options = {})

    => generiert ein Textfeld für eine Instanzvariable mit der Bezeichnung *object_name*
    dem referenzierten Objekt wird eine Nachricht *method* gesendet
    die Rückgabe wird dann in das Textfeld übernommen
- DateHelper
  - enthält Methoden zum Generieren von Select-Boxen zur Darstellung eines Datums
  - teilweise ausgelegt für Attribute eines Modell-Objektes
  - Bsp. **date_select**(object_name, method, options = {})

    => generiert drei Select-Boxen (Jahr, Monat, Tag) für eine Instanzvariable *object_name*
    dem referenzierten Objekt wird eine Nachricht *method* gesendet
    die Rückgabe muss ein Date-Objekt sein und wird zur Vorbelegung genutzt
- URLHelper
  - bietet Methoden um einfach Links, Buttons zu realisieren, die zu einem bestimmten Controller bzw. einer bestimmten Action führen sollen
  - Bsp. **button_to**(name, options = {}, html_options = {})

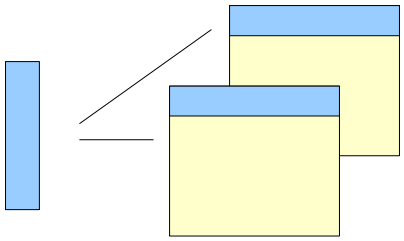
    => generiert einen Button mit dem Value-Attribut *name* die *options* HashMap enthält die Deklaration des Controllers und/oder der Action

Neben den nützlichen Unterstützung auf Code-Ebene der Views bleibt Rails auch hier dem Grundprinzip DRY treu. Bereiche wie etwa Header, Footer usw. können über Layouts ausgelagert werden. Anschließend wrappt ein solches Layout eine zu rendernden View. Über das Schlüsselwort *yield* wird im Layout bestimmt an welcher Stelle die View in das Layout injiziert werden soll.



Layout mit einem Header - der Content ist ein View

Falls darüber hinaus Code-Stücke im RHTML-Code identifiziert werden, die an unterschiedlichen Stellen gebraucht werden, können diese in sogenannte *Partials* gepackt werden. Diese besonderen Views können dann explizit in Views eingebunden werden.



Schematische Funktionsweise von Partials

Auch aktuelle Trends sind an Rails nicht vorbeigegangen und somit findet auch das Thema Web 2.0 bzw. Ajax im Framework Berücksichtigung. Rails bietet dabei u.a. ein leichte Einbindung der Prototype JavaScript Bibliothek. Helper-Klassen nehmen dem Entwickler darüber hinaus viel Arbeit ab und ersparen insbesondere auch den direkten Kontakt mit JavaScript. Zu den gebotenen Funktionalitäten des Frameworks zählen u.a. :

- asynchrone Requests (Updates von HTML-Elementen)
- Effekte (togglen, sliden von Elementen)
- Drag'n Drop Unterstützung
- Autocompletation

2.5 Generatoren

Rails ist so konzipiert, dass redundante Tätigkeiten möglichst vermieden werden. Das Framework und kleine integrierte Tools unterstützen den Entwickler bei seinen Aufgaben, wo es nur geht. Einige dieser wiederkehrenden Aufgaben übernehmen Generatoren. Sie helfen dabei Grundgerüste zu erzeugen. Zur Auswahl stehen u.a. Generatoren zum Generieren

- der Projektstruktur,
- eines Models,
- eines Controller und
- eines Scaffold.

Scaffolding bezeichnet einen besonders interessanten Generator. Dieser erzeugt für ein gegebenen Modellnamen ein Grundgerüst, dass Create, Read, Update und Delete Aktionen (CRUD) beherrscht und eine simple Oberfläche bereitstellt. Im Detail werden dabei ein leeres Model, ein Controller mit den entsprechenden CRUD-Actions respektive der Interaktion mit dem Model, sowie ein Standard-Layout und einfache Views für die korrespondierenden Actions im Controller erzeugt.

Mit Hilfe des Scaffolding lässt sich so mit wenig Aufwand ein erster Prototyp erstellen, der auch als Ausgangspunkt für die Weiterentwicklung der Anwendung dienen kann.

2.6 Beyond

Analog zur Einführung in Ruby kann natürlich auch an dieser Stelle nicht alles zum Thema Ruby on Rails behandelt und bis ins letzte Detail beleuchtet werden. Deshalb folgt an dieser Stelle nun wiederum ein kleiner Ausblick auf weitere interessante Aspekte und Eigenschaften:

Umgebungen

- In Ruby stehen die drei Umgebungen *Development*, *Production* und *Test* zur Verfügung. Jede dieser kann unterschiedlich konfiguriert werden. Manipulierbare Einstellungen betreffen u.a. das Caching-Verhalten, das Class-Loading, das Logging und insbesondere auch die Angabe einer Datenbank-Verbindungen pro Umgebung. Die Test-Datenbank wird dabei bei jedem neuen Test gelöscht und neu angelegt, um Reproduzierbarkeit sicherzustellen.

Tests

- Eine Testunterstützung darf in Rails natürlich auch nicht fehlen. Die Tests sind dabei unterteilt in Unit-Tests (Modell), funktionale Tests (Controller) und Integrationstests (Controller, Models). Die Test-Architektur aller Bereiche ähnelt dabei sehr dem von Junit her bekannten Konzept. So gibt es Methoden zum Initialisieren und Aufräumen von Abhängigkeiten oder benötigten Ressourcen, genauso wie verschiedenste *asserts*, mit Hilfe derer man ein erwartetes Ergebnis abfragen kann. Ebenso unterscheidet auch Unit-Testing in Ruby zwischen erwarteten Fehlern (*failures* durch *asserts*) und unerwarteten Fehlern *(errors* die zur Laufzeit auftreten*)*. Darüber hinaus bietet Rails mit den *Fixtures* ein spezielles Konzept, um die Datenbank mit Test-Daten zu füllen. Über ein spezielles Format können dabei Modelldaten benannt und spezifiziert werden. Für jeden Test (damit ist hier der Aufruf einer Testmethode gemeint) werden die Daten neu in die Test-Datenbank geschrieben und können über einen zuvor gewählten Bezeichner direkt in Tests verwendet werden.

Logging

- Gleichwohl wie beim Testen muss auch Logging-Funktionalität nicht über externe Projekte „eingekauft“ werden. Rails bringt auch hier alles Nötige mit. So werden alle Http-Request und auch vom O/R Mapper getätigte Datenbank-Abfragen automatisch geloggt. Außerdem kann die API natürlich auch zum Loggen eigener Anliegen innerhalb eines Controllers bzw. Models benutzt werden.

Server

- Das Deployment und das Bändigen eines Servers sind Punkte, die bei anderen Frameworks oft Kopfschmerzen bereiten. Benötigt man keinen extrem leistungsstarken Server kommt man mit Rails ohne jegliche Konfiguration und Deployment aus. Der Grund dafür ist der in jede Rails Anwendung integrierte Web-Rick Server. Dieser lässt sich leicht mit einem Einzeiler in der Kommandozeile starten und kann mit einer der Umgebungen konfiguriert werden. Für gehobene Ansprüche besteht zudem u.a. die Möglichkeit Rails per CGI im Apachen anzusprechen.

Mails

- Rails ermöglicht nicht nur das Behandeln von Http-Requests mit Controllern, auch das Senden und Empfangen von Mails stellt für Rails kein Problem dar. Dafür werden lediglich Modelle benötigt die - statt vom O/R Mapper - von *ActionMailer::Base* erben. Wie bei Controllern gibt es zu jedem Mail-Model eine korrespondierende View, die ein Template für Mails darstellt. Für das Senden von Mails arbeitet Rails u.a. mit dem Unix-MTA (MailTransferAgent) *sendmail* zusammen oder spricht auf Wunsch auch das SMTP. Beim Empfangen hat man dann z.B. die Wahl zwischen den MTAs *sendmail* und *postfix*.

Web-Services

- Neben dem *ActionMailer* für E-Mails offeriert Rails auch die Möglichkeit Web-Services zu Konsumieren und Anzubieten. In einem *ActionWebService* werden dafür die Dienste in Form von Methoden implementiert. Die Schnittstelle des WebService wird dann über eine explizite Klasse deklariert *(WebServiceApi)*. Auf der technischen Seite steht für die eigentlichen, entfernten Aufrufe die Kommunikation per SOAP oder XML-RPCs zur Auswahl.

3 Fazit

Lässt man alle Features und Eigenschaften von Ruby on Rails Revue passieren, kann man sich gut vorstellen, dass die Entwicklung von Web-Anwendungen mit diesem Framework gegenüber einigen Anderen viele Vorteile in sich birgt und auf eine effizientere Art und Weise vonstatten geht. Besonders (be)merkenswert ist vor allem die geringe Zeit, die für die Konfiguration benötigt wird. Während man bei anderen Technolgien noch an XML-Deskriptoren schreibt, implementiert man mit Rails schon Anwendungslogik

Neben den vielen Konventionen, die ein schnelleres Entwickeln erlauben, trägt vor allem die Homogenität des Frameworks zur Steigerung der Effektivität bei. Der Entwickler muss nicht verschiedenste Technolgien/Projekte benutzen, konfigurieren und zum Interagieren nötigen. Stattdessen integriert Rails alle nötigen Aspekte, um ein Web-Anwendung realisieren zu können von Haus aus. Rails bietet sozusagen eine „All in One“-Lösung mit O/R Mapper, Logging und Tests usw. .

Natürlich ist aber auch Ruby on Rails nicht die sagenumwobene „Eierlegende Wollmilchsau“ auf dem Gebiet der Web-Anwendungen. So bietet Java aufgrund der gewachsenen Gemeinde und der größeren Popularität ausgereiftere und vielschichtigere APIs, sowie eine bessere Tool-Unterstützung. Und auch auf dem Gebiet der Skalierbarkeit und der Anwendbarkeit in sehr großen Projekten, sind noch einige Fragen offen. Nichtsdestotrotz ist Ruby on Rails mehr als nur eine Alternative zu den etablierten Technologien. Besonders wenn die zu entwickelnde Anwendung viele CRUD-Operationen benötigt, kann über Rails' Generatoren in sehr kurzer Zeit ein funktionsfähiges Programm gebaut werden. Und natürlich eignet sich RoR aufgrund seiner Eigenschaften auch hervorragend für Rapid Prototyping. Mit Ruby baut das Framework dabei auf eine elegante und mächtige Sprache. Die skripttypischen fehlenden Typen sind für Java-Entwickler zwar sehr gewöhnungsbedürftig, jedoch bietet die Dynamik und Funktionalität der Sprache neue und interessante Möglichkeiten bei der Programmierung.

Literatur

- www.rubyonrails.org
- www.rubyonrails.de
- www.ruby-lang.org/en
- www.b-simple.de/documents/download/4
- www.kyleshank.com/files/rails_case_study.pdf
- www.b-simple.de/download/rails-overview.pdf