Datenbankapplikationen mit Ruby on Rails

Datenbanksysteme 2011
Universität Osnabrück
Gastvorlesung von Nicolas Neubauer

Ruby

- plattformunabhängige, höhere *Programmiersprache*
- 1995 erschienen
- Yukihiro Matsumoto, Entwickler
- ~ 2000 Bekanntheit außerhalb Japans
- Objektorientierung
- dynamische Typisierung
- Garbage Collection
- ...

Rails

- Open Source *Webapplication* Framework
- 2004 erschienen
- Anfang 2011 in Version 3

- David Heinemeier Hannsson
- Ziel: schnellere, „agilere“ Webentwicklung
- häufig benötigte Funktionalität bereits vorhanden

Kurzeinführung in Ruby

- „alles ist ein Objekt“
- `42` ist eine konstante Zahl (Fixnum)
- `:symbol` ist ein „konstanter String“
- „`String`“ ist ein String mit Inhalt String
- `meine_variable = 42` ist eine Variablenzuweisung
- Punktsyntax für Methodenaufrufe:
  `„Ruby ist klasse“.sub(„klasse“, „toll“)`

Ruby: Collections

- `[]` ist ein leeres Array `<=> Array.new`
- `[42,23]` ist ein Array mit Einträgen 42 und 23
- `mein_array[0]` gibt den ersten Eintrag zurück
- `{}` ist ein leerer Hash (etwa Hash-Map in Java)
- `{:key => „value“, :key2 => „value2“}`
- `mein_hash[:key]` gibt den Wert für den Schlüssel `:key` zurück

Ruby: Kontrollstrukturen

```
if i == 5
  return „i ist 5“
end

return „i ist 5“ if i == 5
```

```
unless i == 5                    # ~  if !(i == 5)
  return „i ist nicht 5“
end

return „i ist nicht 5“ unless i == 5
```

Ruby: Kontrollstrukturen

```
while i > 0
  i -= 1
end

until i == 0
  i -= 1
end

for element in mein_array
  puts element.to_s
end
```

Ruby: Methoden und Variablen

```
def hallo_welt

  lokale_variable1 = 23 #dynamisch typisiert
  lokale_variable2 = „Hallo Welt!“

  @instanz_variable = „In der Instanz sichtbar!“
  @@klassen_variable = „vgl. static in Java“

  $globale_variable = „überall sichtbar“

end
```

Impliziter Rückgabewert der Methode

`hallo_welt()` ist String: „überall sichtbar“

Ruby: Beispiel - Collatz

```
def collatz(x)

  anz_iterationen = 0

  until x == 1
    if x % 2 == 0
      x = x/2
    else
      x = 3*x+1
    end
    anz_iterationen += 1
  end

end
```

Ruby: Syntax-Besonderheiten

- Weglassen von Klammern bei Eindeutigkeit

```
[42,23].include? 42 <=> [42,23].include?(42)

methode_mit_hash_paramater :key => „value“
                           <=>
methode_mit_hash_paramter({:key => „value“})
```

- Übergeben von Codeblöcken an Methoden

```
10.times do |i|
  puts i
end
```

Ruby: Syntax-Besonderheiten

3 Konzepte von Rails

Don‘t Repeat Yourself

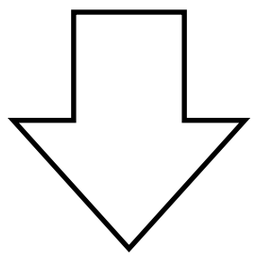
Model-View-Controller

Convention Over Configuration

Model-View-Controller Design Pattern

Idee:

Code nach Kategorien M,V,C strukturieren.



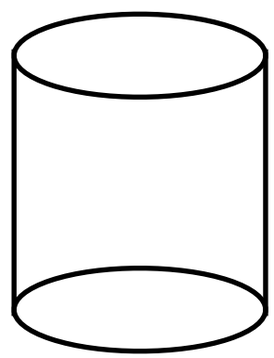
- bessere Programmstruktur
- Wiederverwertbarkeit von Komponenten
- Flexibilität

Model-View-Controller Design Pattern

1. Model und ORM

- Programmteile, die auf Datenhaltung arbeiten
- Abstraktion von der eigentlichen Datenhaltung

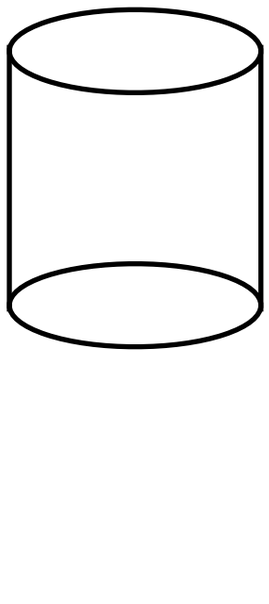
```
class Modelklasse

  def besorge_nachrichten()

  end
```



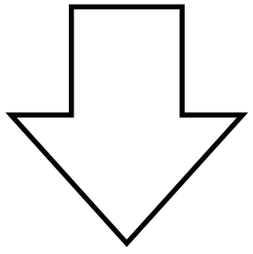
```
Nachrichten:
{[NachrichtID, Nachricht,
     Nutzer, ...]}
```

Model-View-Controller Design Pattern

1. Model und ORM: Objektrelationales Mapping



```
Nachricht: {[NachrichtID, Nachricht,
Nutzer, ...]}
```




```
class Nachricht

end
```

Model-View-Controller Design Pattern

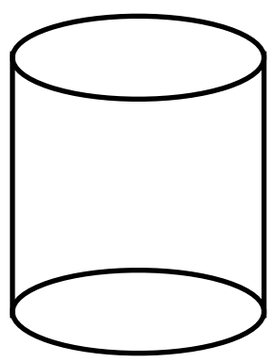
1. Model und ORM: Objektrelationales Mapping

`Nachricht.all`

übersetze Tupel-Ergebnisse in Objekte

übersetze in SQL

`SELECT * FROM Nachricht`

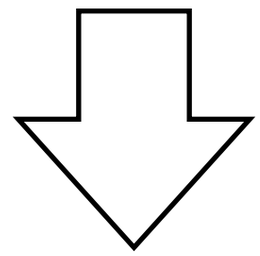


Model-View-Controller Design Pattern

2. View

- (grafische) Datenrepräsentation
- Templates

```
<li><%= @nachricht %></li>
```



```
<li>Hallo Welt!</li>
```

Model-View-Controller Design Pattern

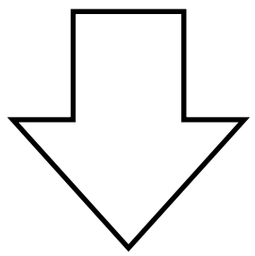
3. Controller

- Applikationssteuerung
- Vermitteln zwischen Daten und Repräsentation

Model-View-Controller Design Pattern

3. Controller

```
zeige_alle_nachrichten()
```

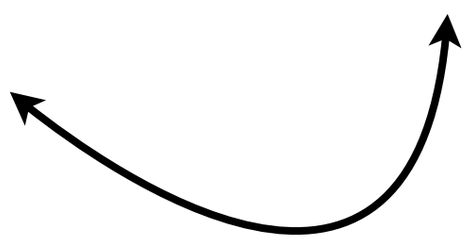


Controller-Klasse

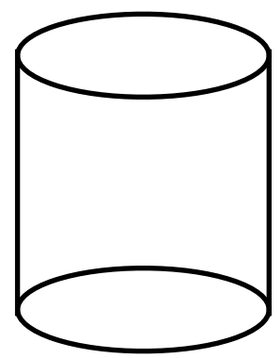
```
Model.besorge_alle_Nachrichten()
View.zeige_Template_mit_
  Nachrichten(alle_Nachrichten)
```

Template

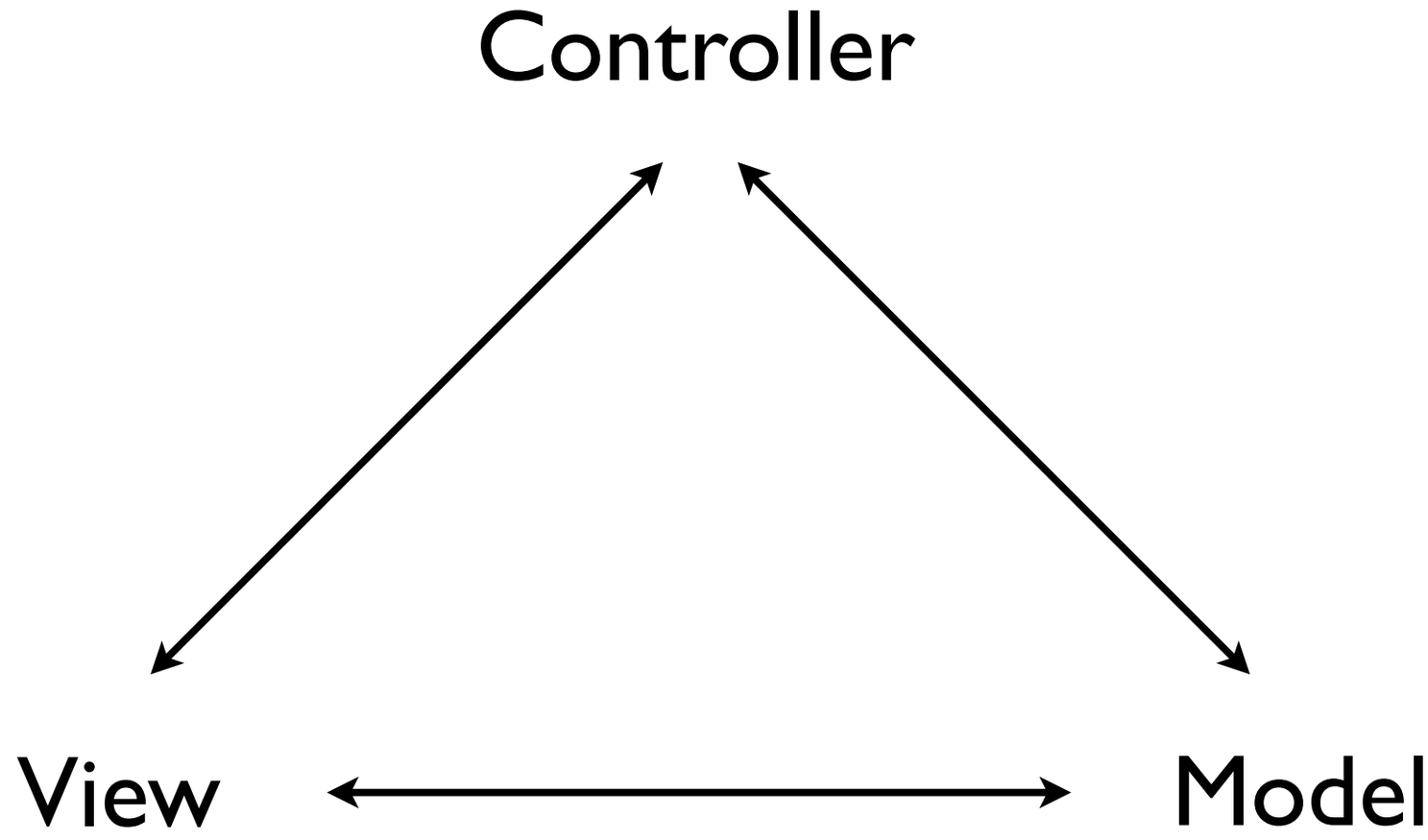
```
<html>
  ...
</html>
```



Modell Klasse



Model-View-Controller Design Pattern



Eine Beispielapplikation

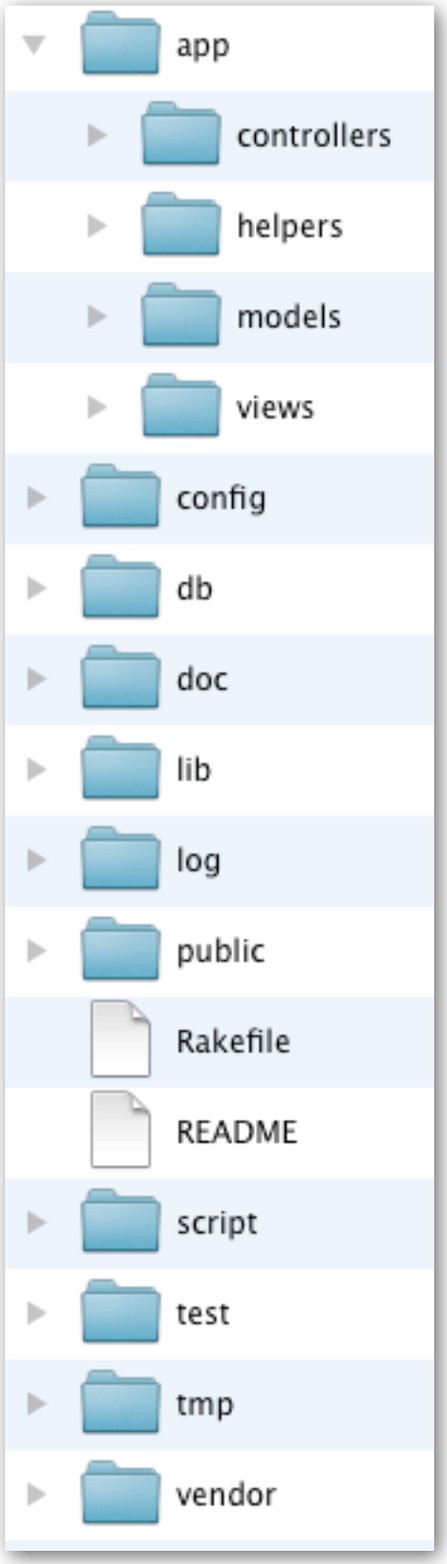
neue Rails Applikation erzeugen:

```
$ rails new meine_applikation -d mysql
```

Applikationsname

Datenbanktyp im Backend,
SQLite: Standard

Eine Beispielapplikation



eigentliche Applikationsdaten

Konfigurationsdateien,

Datenschemadefinitionen, „Migrations“

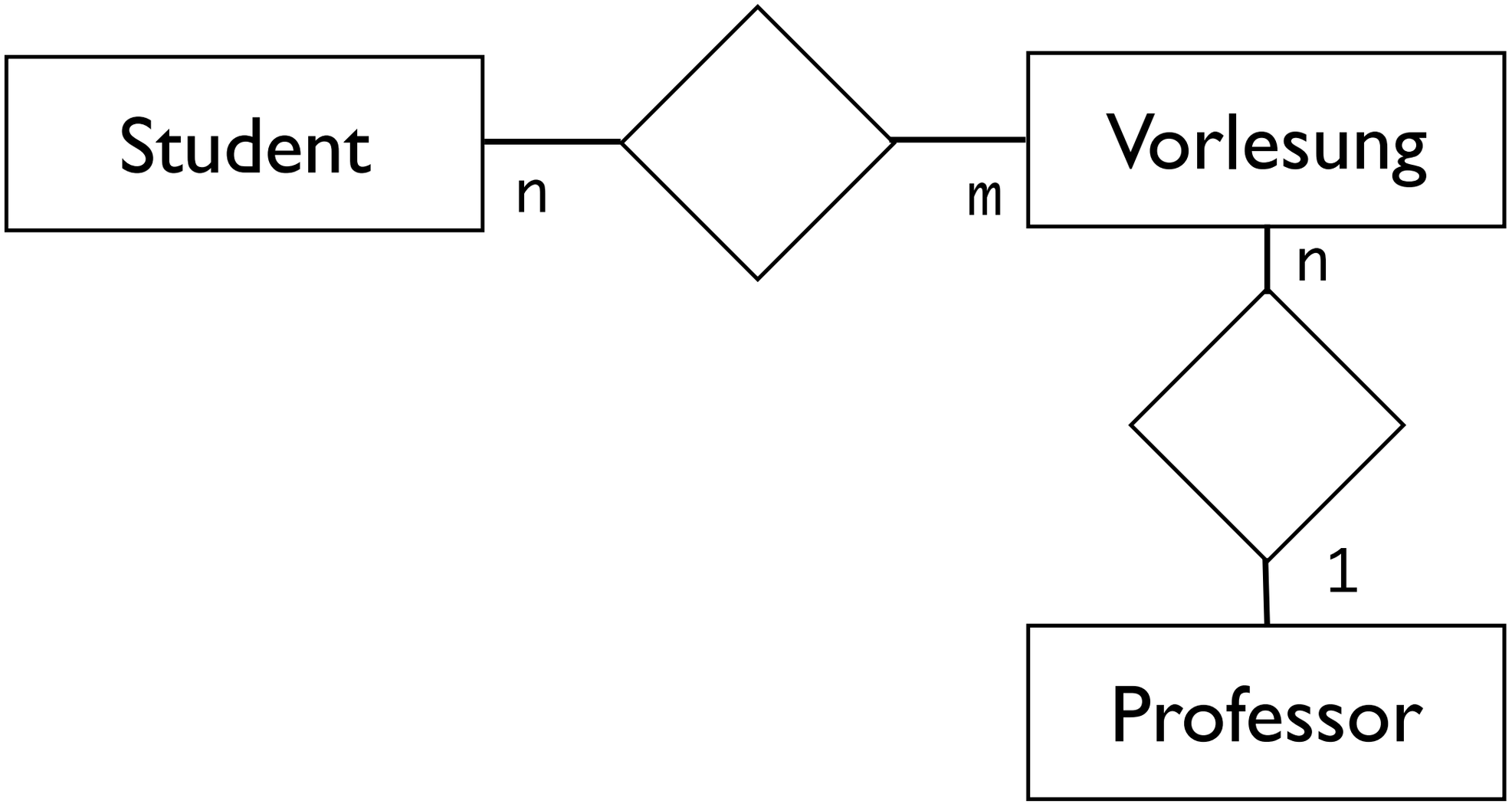
Testumgebung

Eine Beispielapplikation: Datenbankverbindung

- Rails mit Datenbank verbinden
- YAML-Konfigrationsdatei:
  `config/database.yml`

```
development:
adapter: mysql
encoding: utf8
reconnect: false
database: DATENBANKNAME
pool: 5
username: BENUTZERNAME
password: PASSWORT
host: HOSTNAME
```

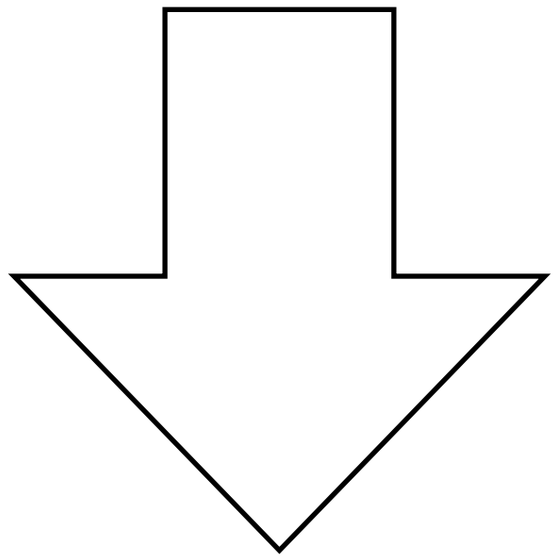
Eine Beispielapplikation



Convention over Configuration

Idee:

Halte dich an bestimmte Bedingungen ...



... erhalte Funktionalität ohne Konfigurationsaufwand.

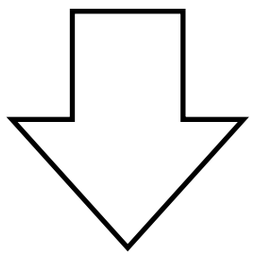
Namenskonventionen

- Beispiel: Entity-Typ „Vorlesung“
- Tabellenname *underscored* im Plural:
  `vorlesungen` (`informatik_vorlesungen`)
- *Model-Klasse* im *Mixed Case*:
  `Vorlesung` (`InformatikVorlesung`)
- Obacht: englische Pluralbildung:
  `Vorlesung` **wird** `Vorlesungs`
  (**vgl.** `Lecture`, `Lectures`)
- Also: Entitäten in Englisch benennen

Schlüsselkonventionen

- Schlüsselkonvention:
  synthetischer Schlüssel „id“ für jede Entität

```
Vorlesung: {[VorlNr, Titel, PersNr, ...]}
```

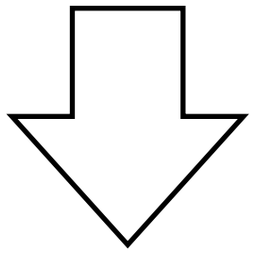


```
lectures: {[id, vorl_nr, titel, pers_nr, ...]}
```

Schlüsselkonventionen

- Konvention für Fremdschlüsselbeziehungen

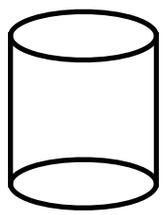
```
Vorlesung: {[VorlNr, Titel, PersNr, ...]}
```



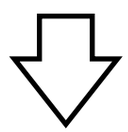
```
lectures: {[id, vorl_nr, titel, professor_id, ...]}
```

Model-Generierung

- Model-Klassen aus Datenbankschema ableiten



```
lectures: {[id, titel, ...]}
```



```
class Lecture < ActiveRecord::Base
...
end
```

```
$ rails generate model lecture
```

Objektrelationales Mapping

- Komfortabler Umgang mit Daten:

```
#Erzeuge neue Vorlesung:
new_vl = Lecture.new

#Fülle Daten
new_vl.titel = „Ruby on Rails“
new_vl.sws = 4
new_vl.professor_id = 1

#Speichern
new_vl.save
```

Objektrelationales Mapping

- Komfortabler Umgang mit Daten:

```
SHOW COLUMNS FROM lectures,
synthetisiere Setter und Getter für alle
Attribute

nutze Setter für gefundenes Attr. titel
nutze Setter für gefundenes Attr. sws
nutze Setter für gefundenes Attr. professor_id


INSERT INTO lectures (...) VALUES (...)
```

Don‘t Repeat Yourself

- Woher weiß Rails, welche Attribute ein bestimmtes Objekt hat?
- Attribute stehen bereits in der Datenbank

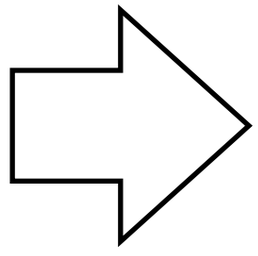
	Feld	Typ	Kollation	Attribute	Null	Standard
☐	**id**	int(11)			Nein	*Kein*
☐	**vorl_nr**	int(11)			Nein	*Kein*
☐	**titel**	varchar(20)	latin1_swedish_ci		Ja	*NULL*
☐	**sws**	int(11)			Ja	*NULL*
☐	**professor_id**	int(11)			Nein	*Kein*

- „Wiederhole dich nicht selbst!“

Datenbankabstraktion, ORM

- CRUD-Operationen mit Klassen/Objekten kapseln
- CREATE (INSERT), READ (SELECT), UPDATE, DELETE

```
my_vl = Lecture.new
my_vl = Lecture.first
my_vl.save
my_vl.delete
```



```
INSERT INTO lectures ...
SELECT * FROM lectures
UPDATE lectures SET ...
DELETE FROM lectures ...
```

- Zur Erinnerung:
  Funktionalität „umsonst“ wegen Konventionen

Datenbankabstraktion, ORM

- Komfortabler Umgang mit Daten:

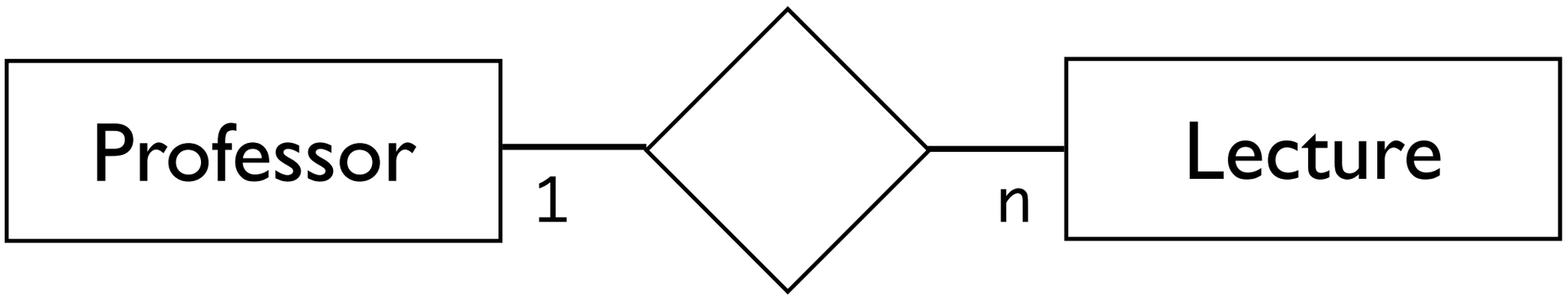
```
#Besorge eine Vorlesung:
vl = Lecture.first

#Welcher Professor hält die Vorlesung
vl.professor


#=> <Objekt vom Typ Professor>
```

Datenbankabstraktion, ORM

- Komfortabler Umgang mit Daten:

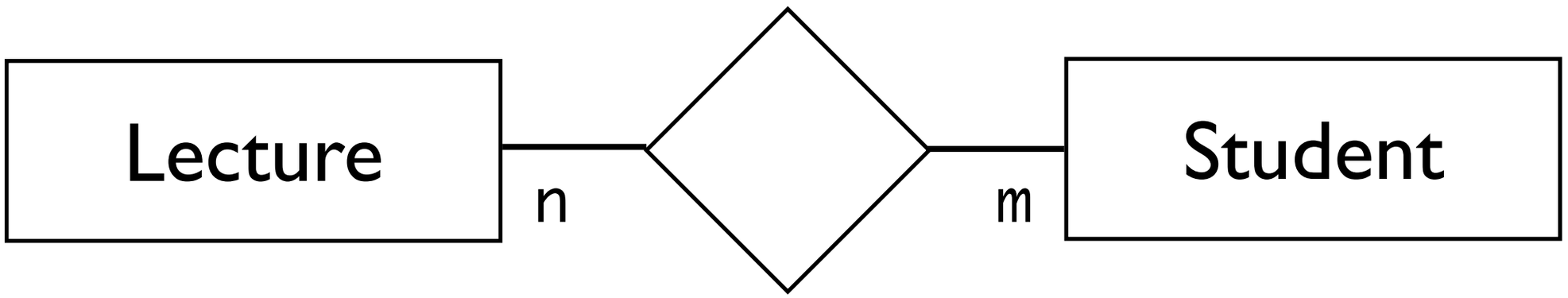
```
SELECT * FROM professors WHERE
id = [vl.professor_id],
Erzeuge neues Professor-Objekt aus dem
Ergebnis-Tupel
```

Model und Beziehungen




```
class Professor < ActiveRecord::Base
  has_many :lectures
end

class Lecture < ActiveRecord::Base
  belongs_to :professor
end
```

Model und Beziehungen




```
class Lecture < ActiveRecord::Base
  has_many_and_belongs_to_many :students
end

class Student < ActiveRecord::Base
  has_many_and_belongs_to_many :lectures
end
```

Model und Beziehungen

- n:m-Relationen-Modellierung in der Datenbank
- nicht zu „verfeinern“

```
lectures: {[id,titel,...]}
students: {[id,matr_nr,...]}
```

- „Join“-Table nötig, Konvention:

```
entität1_entität2: {[entität1_id, entität2_id]}
lectures_students: {[lecture_id, student_id]}
```

- Obacht: L < S, daher *nicht* students_lectures

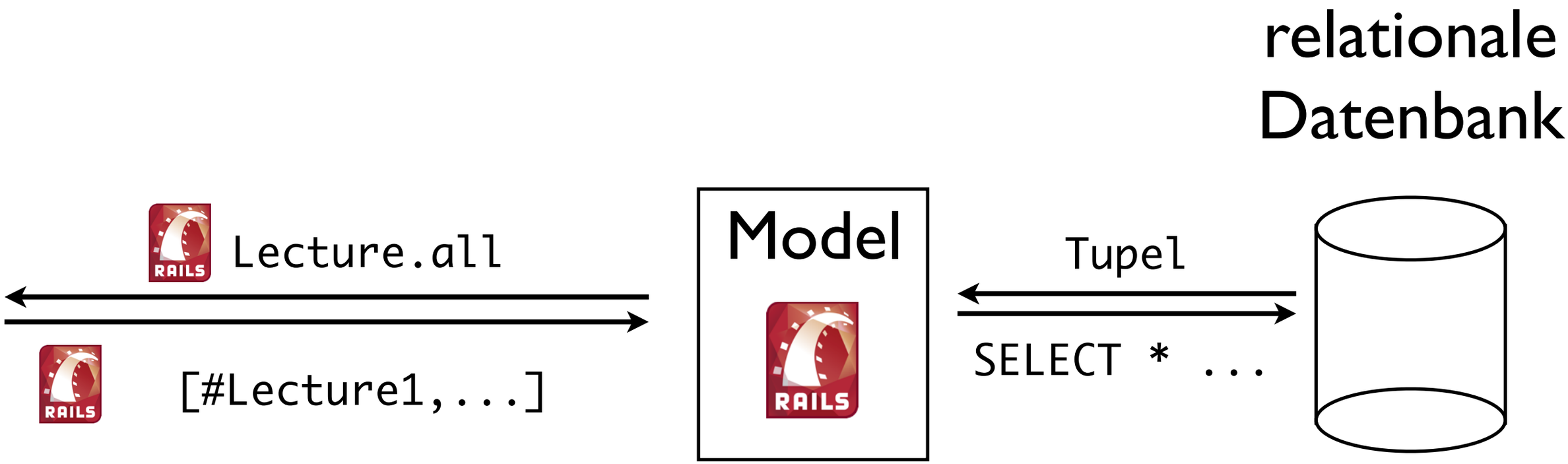
Model und Beziehungen

- Erweiterungen von Model-Klassen um modelbezogene Logik:

```
#liefert alle Professoren eines Studenten
def professors
  self.lectures.collect { |l| l.professor }
end
```

Datenbankabstraktion - Zusammenfassung

- M in MVC - Rails Models
- generiert (DRY, Konventionen)
- Kapseln von Datenbankanfragen
- grundlegende CRUD-Operationen „kostenlos“



Datenbankabstraktion - Zusammenfassung

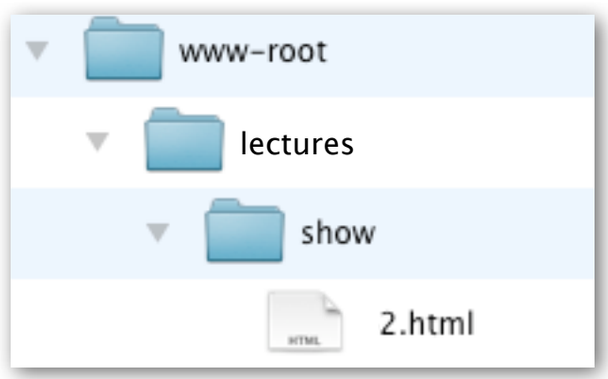
Controller

- C in MCV - Rails Controller
- Rails Applikation aus dem Browser aufrufen

- Klassisch:

  Server: dbs, Port: 3333
  Aufgerufene Seite:

  www-root/lectures/show/2.html



Controller

- C in MCV - Rails Controller
- Rails Applikation aus dem Browser aufrufen

- Rails (Konvention):

dbs:3333/lectures/show/42

Server:Port

Controller

Action

id

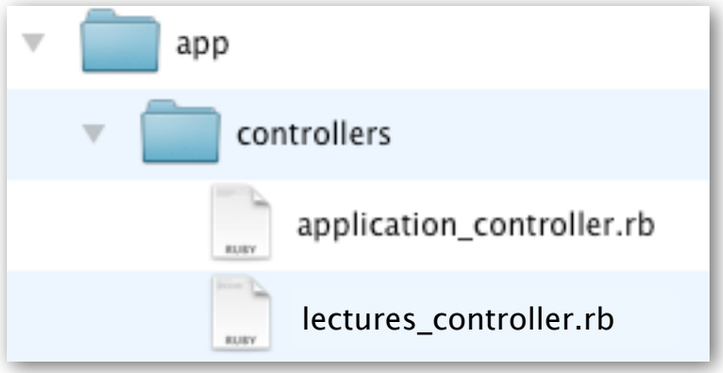
Controller

- 

- Instantiiere die Klasse `LecturesController`
- Rufe die Methode („Action“) `show` auf
- Übergebe der Methode im *Parameter-Hash* für die Variable `id` den Wert `42`

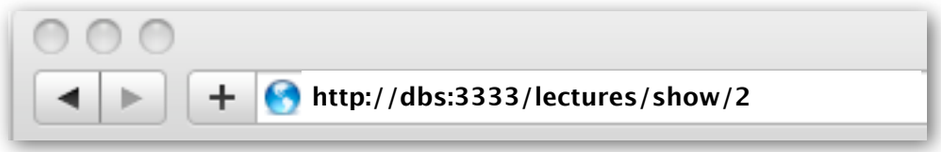
Controller

- Controller erzeugen
- `rails generate controller lectures`




```
class LecturesController < ApplicationController

end
```

Controller




```
class LecturesController < ApplicationController

  def show
    id = params[:id] #enthält Wert 2
    my_vl = Lecture.find(id)
    #findet Lecture mit id 2

  end

end
```

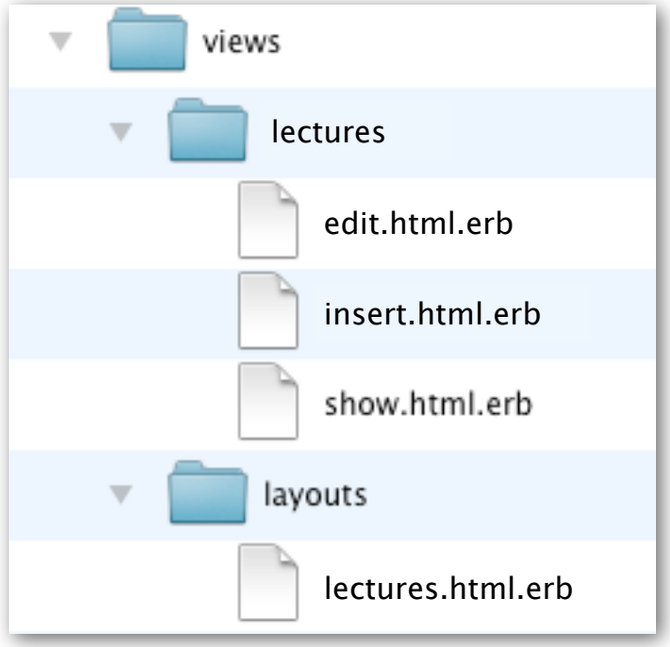
Controller

```
class LecturesController < ApplicationController

  def show

    id = params[:id] #enthält Wert 2
    @my_vl = Lecture.find(id)
    #findet Lecture mit id 2

    render :controller => „lectures“,
           :action => „show“
  end

end
```

Views & Templates

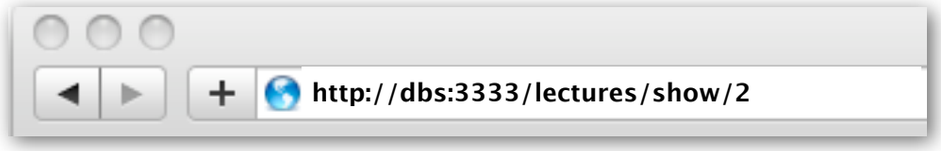
- V in MVC - Rails Views
- Konvention:

  Ordner für Controller
  Templates für Actions
  Layouts für Controller



- Template-Sprache: (Embedded-)Ruby

Views & Templates




```
render :controller => „lectures“,
       :action => „show“,
```

- Verarbeite Template für Action „show“
  (Konvention: `show.html.erb` im Ordner `lectures`)
- Sende Ergebnis an Nutzer zurück

Views & Templates: Embedded Ruby

- Ausgebende und Nicht-Ausgebende Tags

```
<%= „Ich bin ein String!“ %>
<% „Ich bin auch ein String!“ %>
```

- Ausgabe

```
Ich bin ein String!
```

- Sinnvoll bei Blöcken

```
<% if(bedingung) %>
<%= gib_etwas_aus() %>
<% end %>
```

Views & Templates: Zugriff auf Variablen

- Zugriff auf Variablen

```
<%= lokal = @variable_aus_dem_controller %>
<%= lokal %>
```

- Beispiel

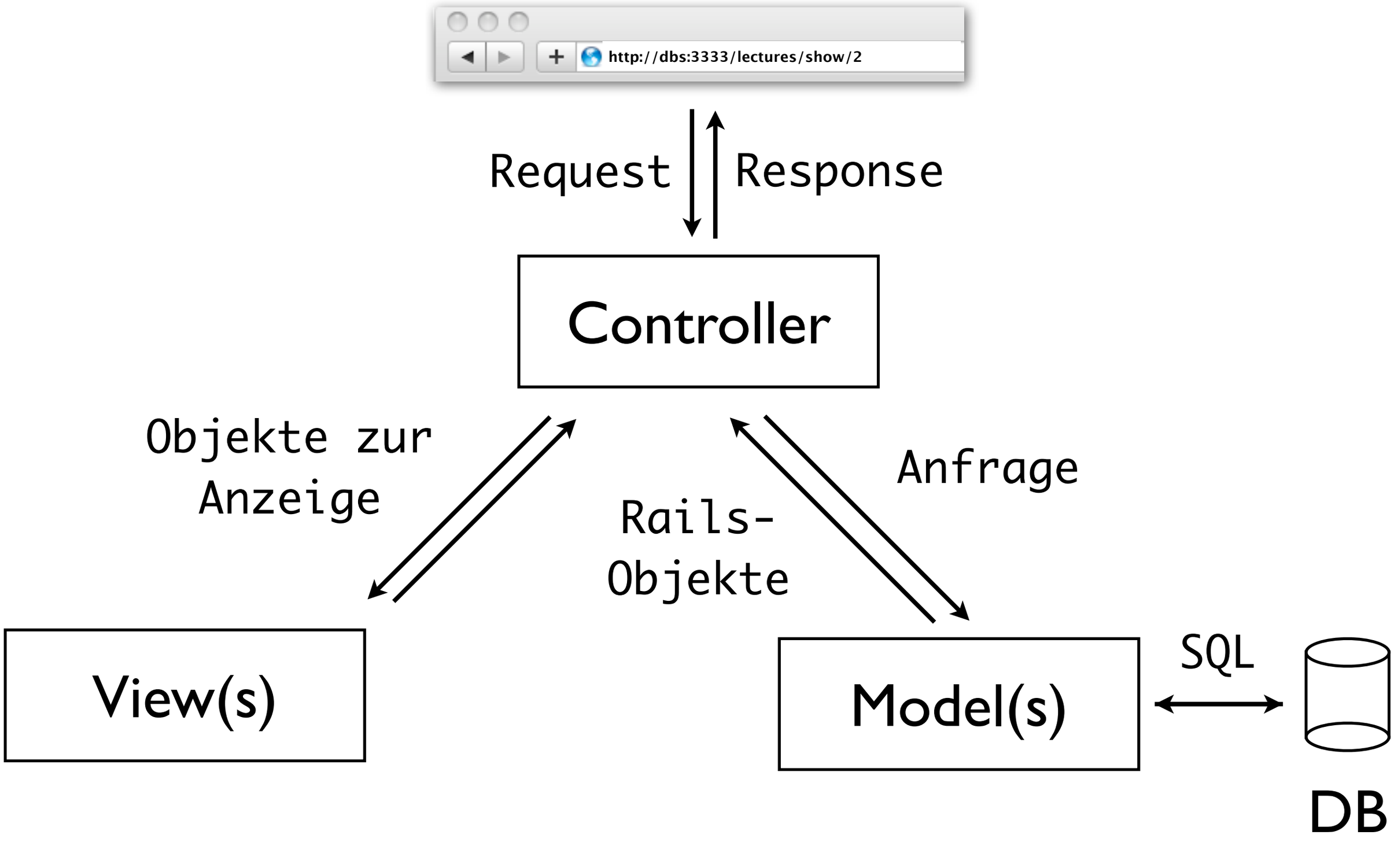
```
<h1>Alle Vorlesungen</h1>
<ul>
<% @lectures.each do |lecture| %>
  <li><%= lecture.title %></li>
<% end %>
</ul>
```

Views & Templates

- Layout beinhaltet Rahmenstruktur
- Platzhalter für Inhalt des eigentlichen Templates

```
<html>
<head>
   <title>Lectures</title>
</head>
<body>

   <%= yield %>

</body>
</html>
```

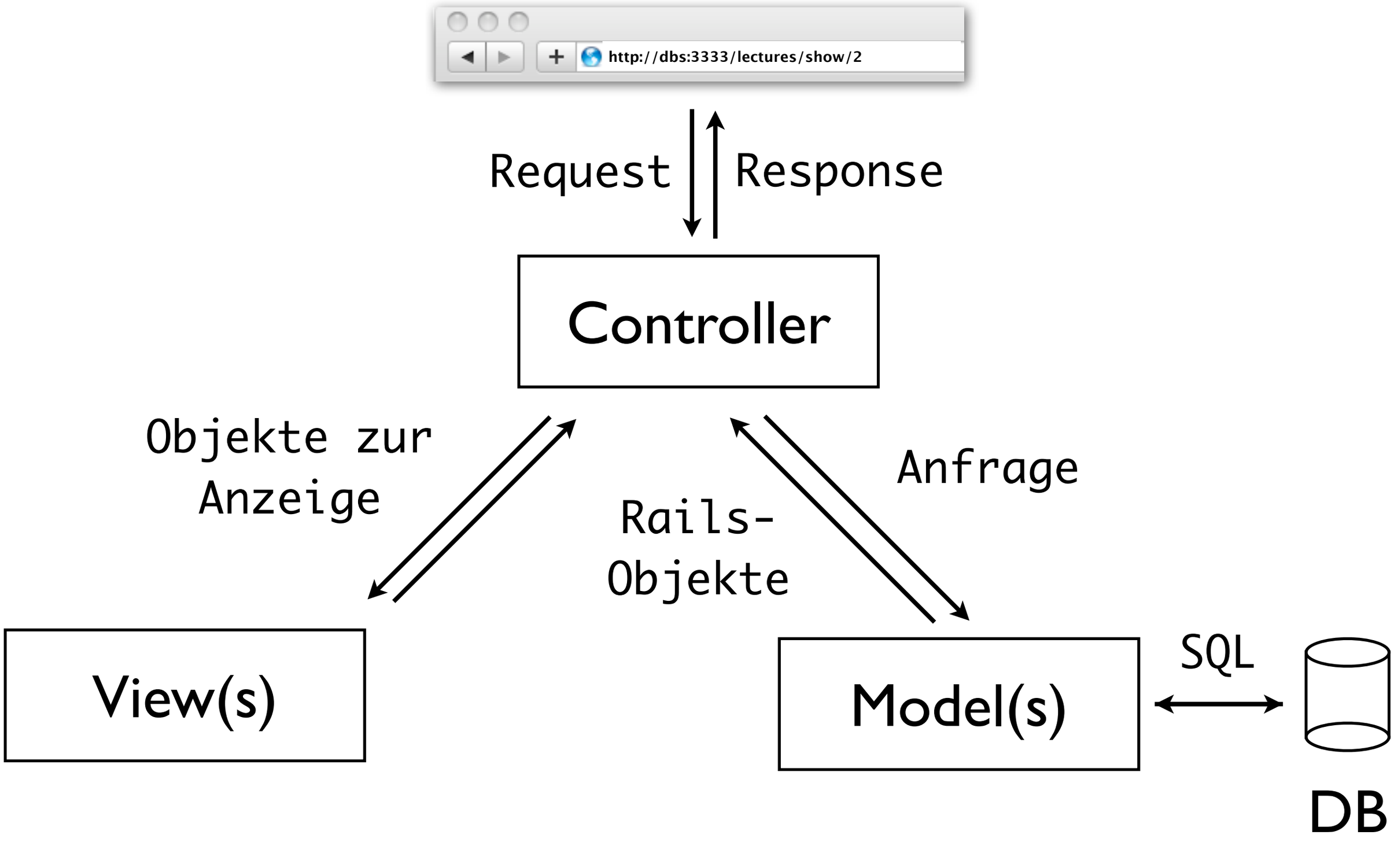
Views & Templates



Datenbankapplikationen mit Ruby on Rails, Teil 2

Datenbanksysteme 2011
Universität Osnabrück
Gastvorlesung von Nicolas Neubauer

Views & Templates



Wiederholung

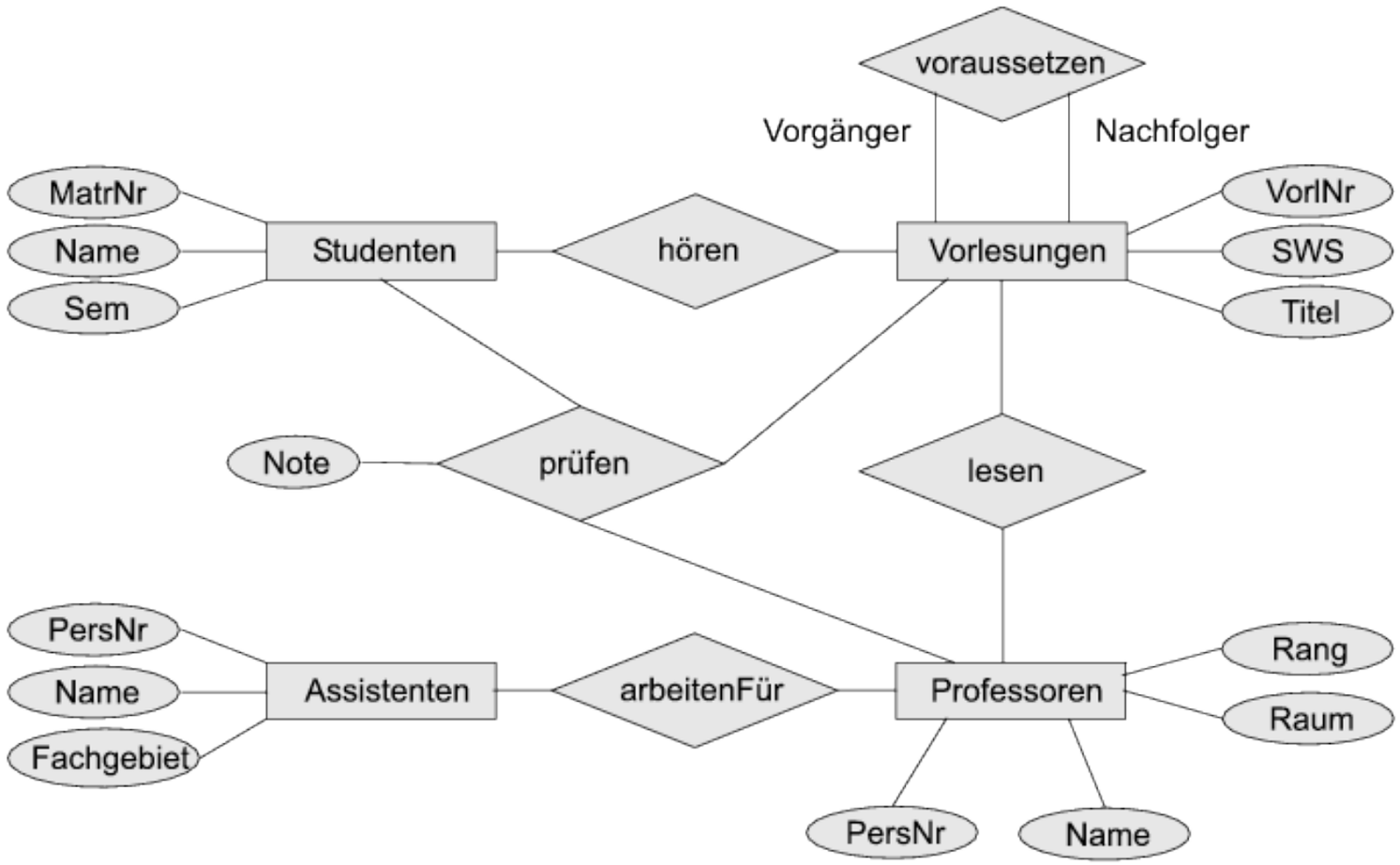
Demo

Wiederholung

- Model-View-Controller
- Don‘t Repeat Yourself
- Convention over Configuration
- Objektrelationales Mapping
- Browser-Request Verarbeitung
- Templates

Neue Applikation von 0

- Bisher: Vorhandene Datenbank
- Jetzt: Komplett neue Applikation



Scaffolds

- Scaffold = Gerüst
- `rails generate scaffold lectures`
- erzeugt:
  - Migration
  - Model
  - Controller
  - Views
  - Tests

Migration

- Methode für iterative Datenbankentwicklung

```
class CreateLectures < ActiveRecord::Migration
  def self.up
    create_table :lectures do |t|
      t.string  :titel
      i.integer :vorl_nr
    end
  end

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

Migration

- Methode für iterative Datenbankentwicklung

```
class AddSWSLectures < ActiveRecord::Migration
  def self.up
    add_column :lectures, :sws, :integer
  end

  def self.down
    remove_column :lectures, :sws
  end
end
```

Migration

- übernimmt später CREATE/ALTER TABLE

```
$ rails generate scaffold lectures
  titel:string sws:integer vorl_nr:integer
  professor:references
```

```
DATENFELD:DATENTYP
```

```
string
text
integer
float
datetime
...
```

Migration

- übernimmt später CREATE/ALTER TABLE

```
$ rails generate scaffold lectures
  titel:string sws:integer vorl_nr:integer
  professor:references
```

<=>

```
$ rails generate scaffold lectures
  titel:string sws:integer vorl_nr:integer
  professor_id:integer
```

Migration

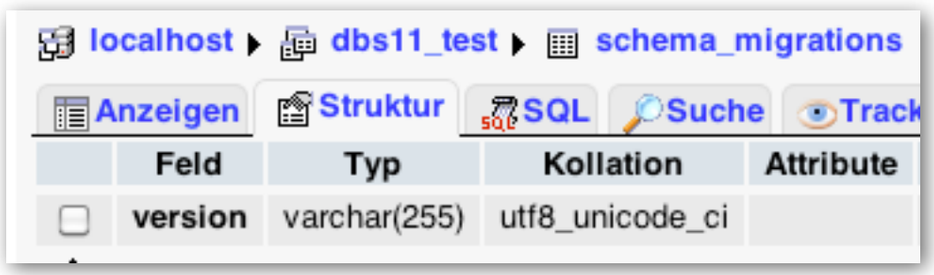
- sind eigene Klassen in

  `meine_applikation/db/migrate`

- können automatisiert ausgeführt werden

  `rake db:migrate VERSION=123`

- Datenbank kennt aktuelle Version



Model

- automatisch generierte Model-Klassen

```
class Lecture < ActiveRecord::Base
  belongs_to :professor
end
```

```
$ rails generate scaffold lectures
  titel:string sws:integer vor_nr:integer
  professor:references
```

- Obacht: Klasse `Professor` wird nicht angepasst

Controller & Views

- Controller und Views für CRUD-Operationen
- betrachtet Entität als Ressource
- REST-Operationen

  GET
  POST
  PUT
  DELETE

- Bietet HTML- und XML-Zugriffsmöglichkeiten

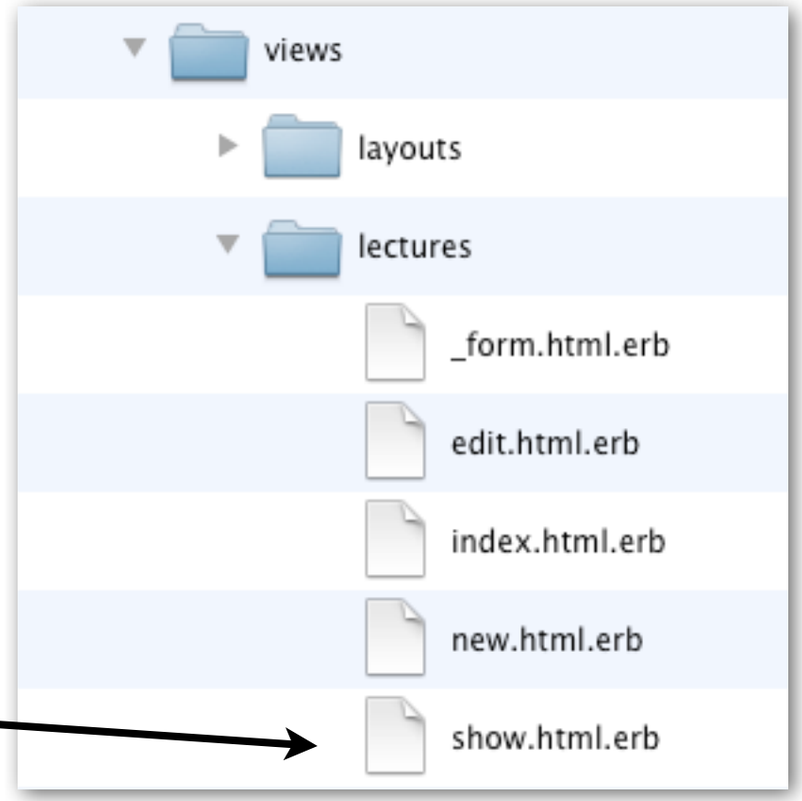
Controller & Views

```
class LecturesController < ApplicationController

def show
    @lecture = Lecture.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @lecture }
    end
  end

  ...

end
```

Controller & Views

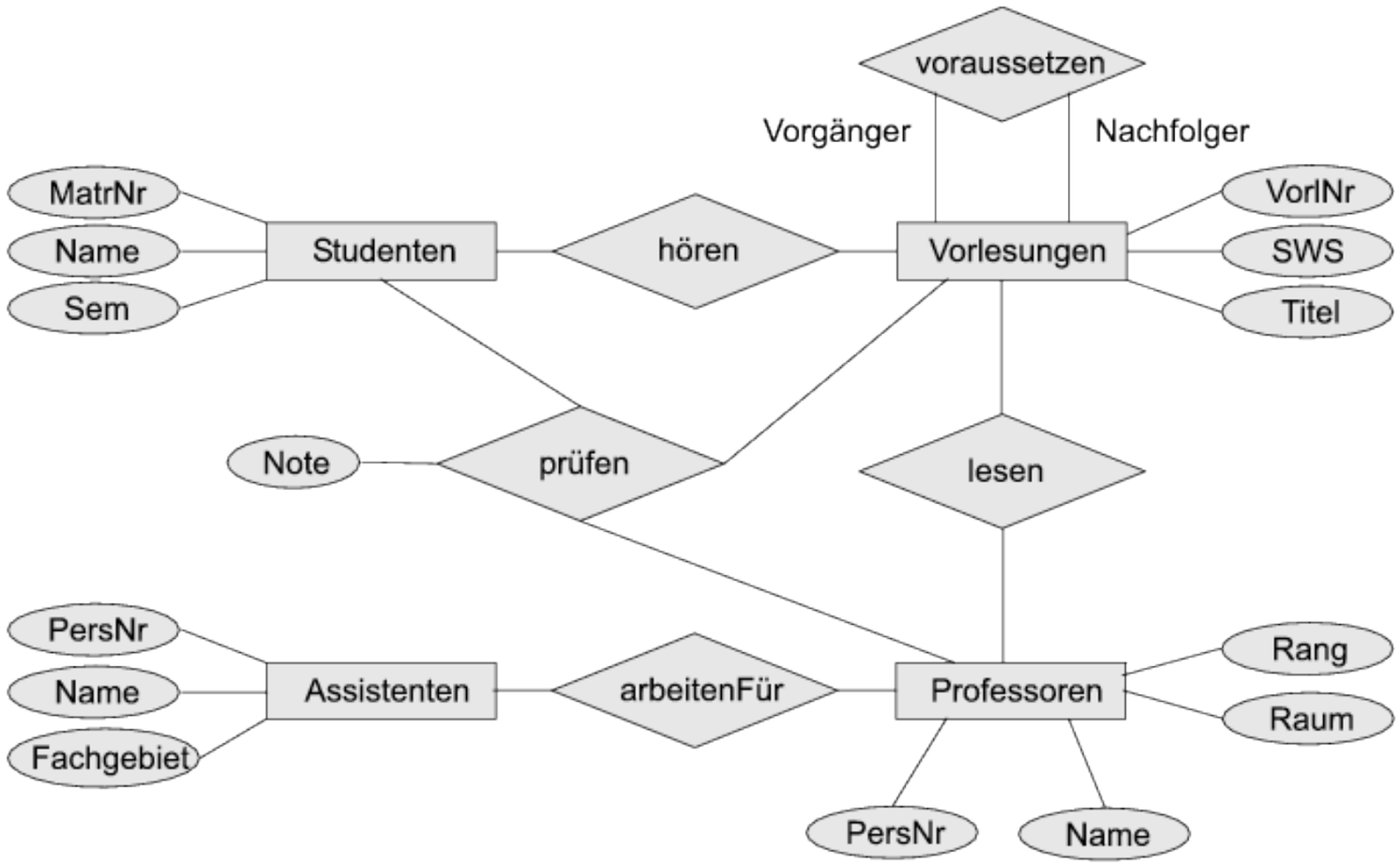
- Erinnerung:

```
def show
   ...
end

render :action => „show“
```



- HTML-Oberflächen für CRUD-Operationen

Neue Applikation von 0

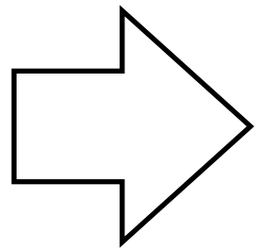


Neue Applikation von 0

Demo

Scaffolds: Wiederholung

- `rails generate scaffold lectures`
- erzeugt:
  - Migration
  - Model
  - Controller
  - Views
  - Tests



CREATE TABLE ...

Lecture.new

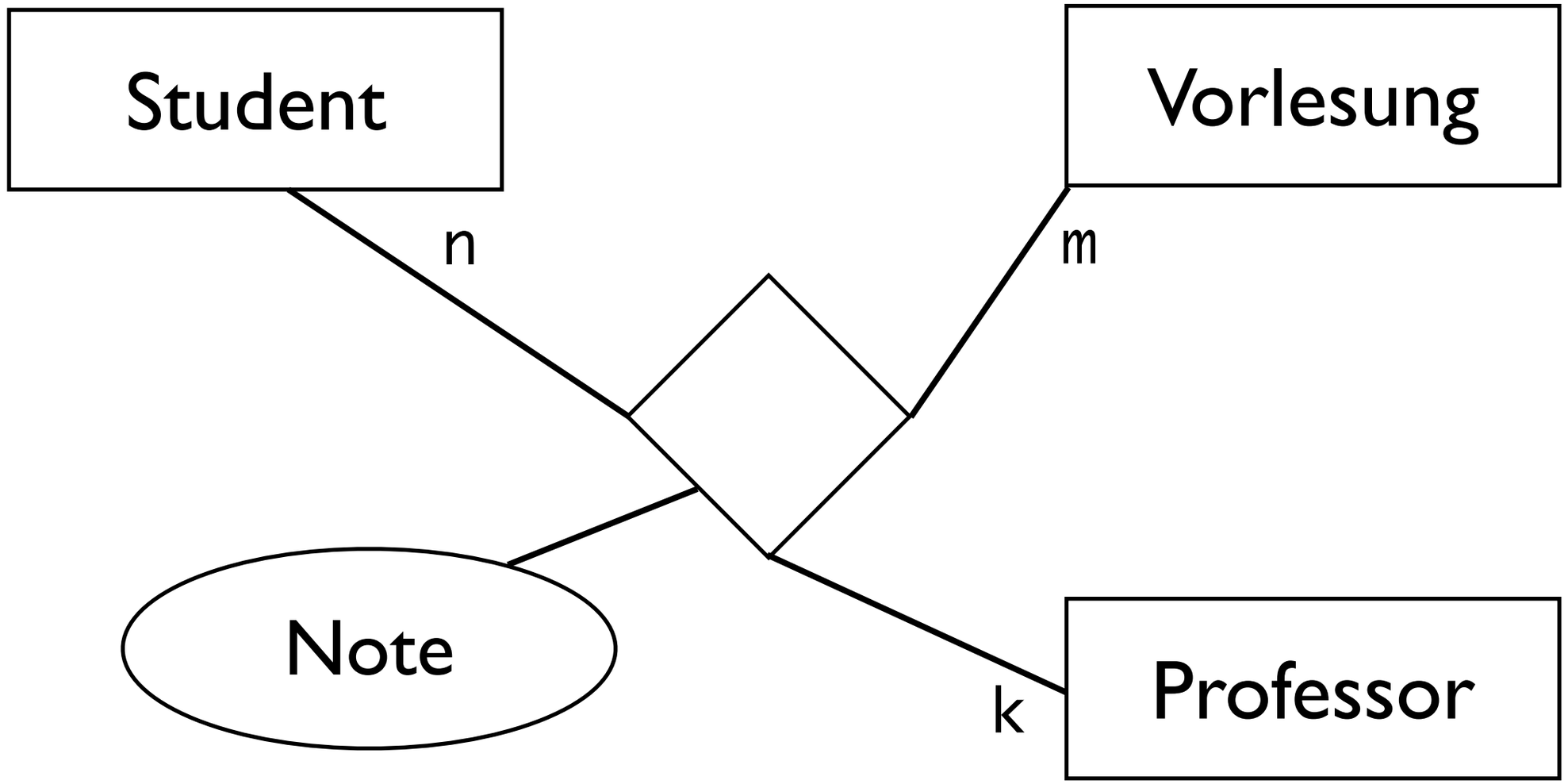
http://.../lecture/show

show.html.erb

...

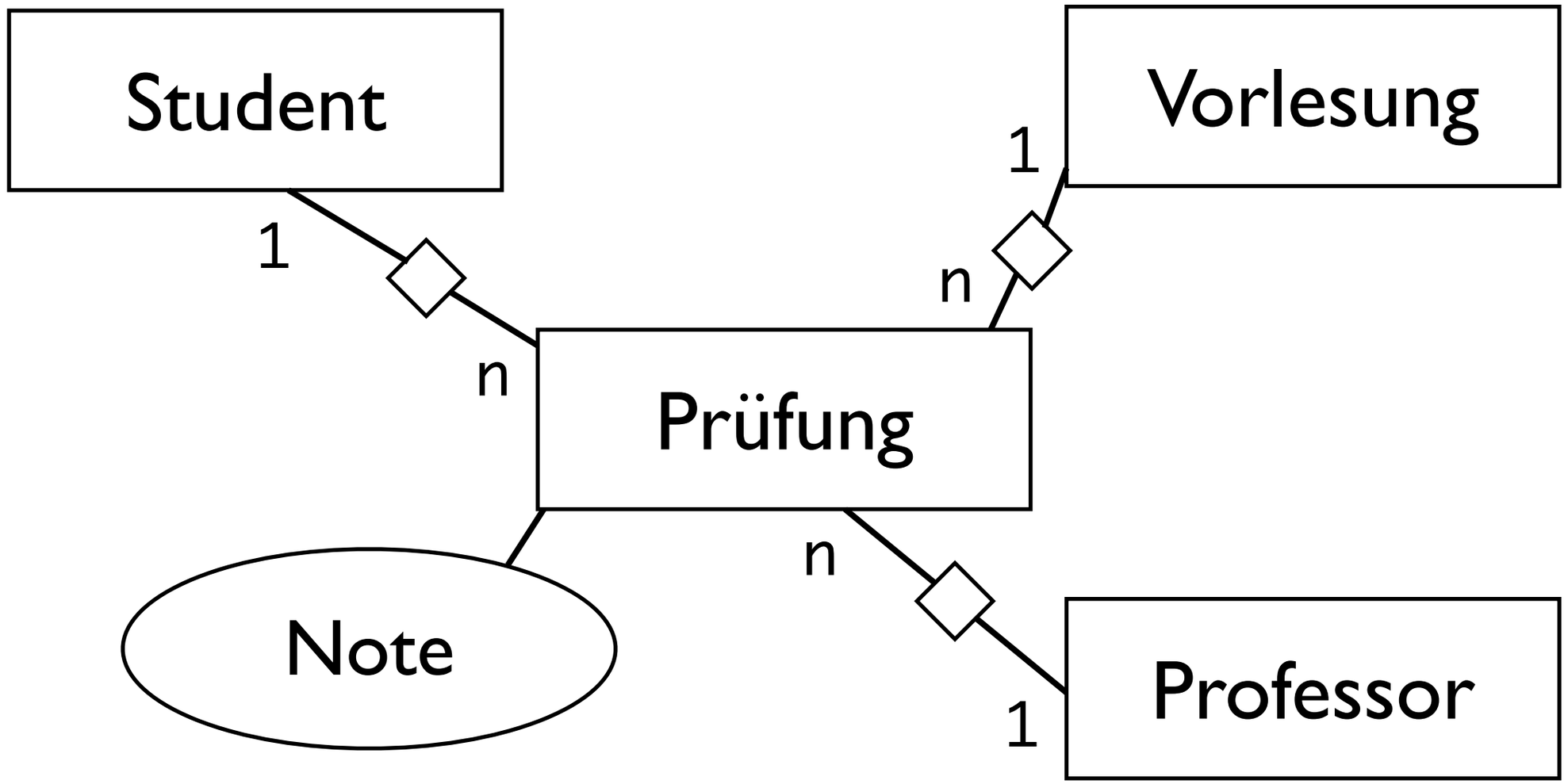
Fortgeschrittene ORM-Techniken

- mehrwertige Relationen und Relationen mit Attributen



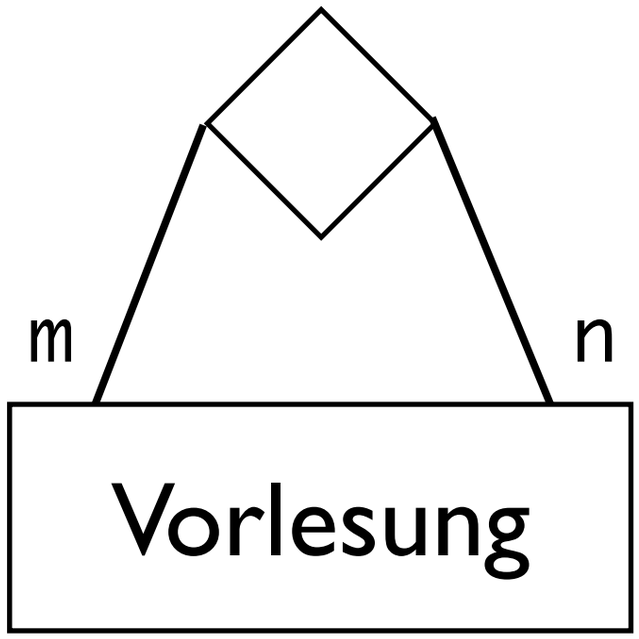
Fortgeschrittene ORM-Techniken

- mehrwertige Relationen und Relationen mit Attributen



Fortgeschrittene ORM-Techniken

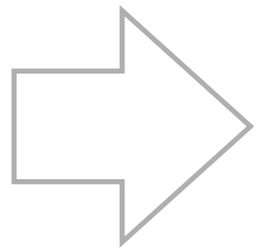
- Entitäten mit N:M-Selbstreferenz




```
has_and_belongs_to_many :vorgaenger,
:class_name => "Lecture", :join_table =>
"lectures_lectures", :foreign_key =>
"nachfolger_id", :association_foreign_key =>
"vorgaenger_id"
```

Scaffolds: Wiederholung

- `rails generate scaffold lectures`
- erzeugt:
  - Migration
  - Model
  - Controller
  - Views
  - Tests



CREATE TABLE ...

Lecture.new

http://.../lecture/show

show.html.erb

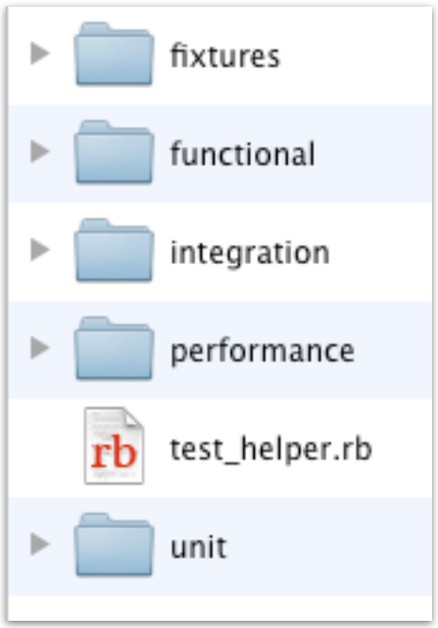
...

Testen in Rails

- Test-Driven-Development
- Kernidee von Rails
- integrierte Tests für
  - Models (Unit-Tests)
  - Controller (Functional Tests)
  - Views (Functional Tests)
  - Controller-Interaktion (Integration Tests)

Testen in Rails

- Verzeichnis `test`



← Testdaten
← Tests für Controller/Views
← Tests für Controller-Interaktion
← Tests für Models

Einfacher Unit-Test

- unit/student_test.yml

```
require 'test_helper'

class StudentTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  test "the truth" do
    assert true
  end
end
```

Einfacher Unit-Test

```
require 'test_helper'

class StudentTest < ActiveSupport::TestCase

  test "valid semester" do
      s = Student.new
      s.name = "Willi"
      s.matr_nr = 123
      s.semester = -1  #fehlerhafte daten
      assert !s.save, "Neg. Semester erlaubt"

  end
end
```

Einfacher Unit-Test

- Test ausführen

```
$ ruby -Itest test/unit/student_test.rb
```

für test_helper.rb

Test(s)

- Lädt Testdaten in Datenbank
- Führt alle Tests aus

Einfacher Unit-Test

- Ergebnis:

```
Loaded suite test/unit/student_test
Started
F
Finished in 0.105289 seconds.

  1) Failure:
test_valid_semester(StudentTest)
[test/unit/student_test.rb:11]:
Neg. Semester erlaubt

1 tests, 1 assertions, 1 failures,
0 errors, 0 skips
```

Test-Driven-Development

- Erst Test schreiben, dann Funktionalität entwickeln
- Beispiel: Validierung von Daten eines Models
- app/models/student.rb

```
validates :semester,
          :presence => true,
          :inclusion => 1..25
```

- Obacht: Niemand verhindert INSERT INTO ...

Einfacher Unit-Test

- Ergebnis:

```
Loaded suite test/unit/student_test
Started
.
Finished in 0.028667 seconds.

1 tests, 1 assertions, 0 failures,
0 errors, 0 skips
```

Fixtures

- Nicht auf Produktivdaten testen
- Fixtures = Testdaten
- YAML oder CSV
- Test-Datenbank in database.yml konfigurieren
- ggf. mit `rake db:test:prepare` initialisieren
- vor dem Testen automatisch in die Testdatenbank geladen

Fixtures

- `fixtures/students.yml`

```
one:
  matr_nr: 1
  name: MyString
  semester: 1

two:
  matr_nr: 1
  name: MyString
  semester: 1
```

Fixtures

- fixtures/students.yml

```
willi:
  matr_nr: 123
  name: Willi Wacker
  semester: 5
  lectures: infob
```

Fixtures

- fixtures/lectures.yml

```
infoa:
  titel: Info A
  vorl_nr: 1
  sws: 9

infob:
  titel: Info B
  vorl_nr: 2
  sws: 9
  vorgaenger: infoa
```

Unit-Test mit Testdaten

- models/student.rb

```
def fehlende_vl

  erg = []
  self.lectures.each do |l|
    erg = erg + (l.vorgaenger - self.lectures)
  end

  return erg.uniq

end
```

Unit-Test mit Testdaten

```
require 'test_helper'

class StudentTest < ActiveSupport::TestCase

  test "vl_ohne_vorgaenger funktion" do
        s = students(:willi)
        assert s.fehlende_vl.include?
               (lectures(:infoa)),
               "Willi sollte Info A fehlen"

  end
end
```

Unit-Test mit Testdaten

- Ergebnis:

```
Loaded suite test/unit/student_test
Started
..
Finished in 0.097540 seconds.

2 tests, 2 assertions, 0 failures,
0 errors, 0 skips
```

Weitere Informationen

- Mehr zum Thema *Testen mit Rails*
- Übungsblatt und Übung
- http://guides.rubyonrails.org/testing.html

Ein „echtes“ Beispiel

- Was fehlt eigentlich noch?
- „echte“ Queries
- Datenbank-Performance ausnutzen

- IMDB-Datenbank:
  O(n * log n) bei 60 Millionen Einträgen...
- ActiveRecord-Query-Interface

ActiveRecord

- Datenabfragen mit eigenem Query-Interface

```
Movie.first
Movie.find(123)
Movie.where(:name => „abc“).where(:year => 1942)
Movie.where(„name = ?“, @name)
Movie.order(„name“)

offset, limit, group, having, ...
```

- Benötigt passende Model-Klasse

ActiveRecord

- Direkte Anfragen an die Datenbank

```
ActiveRecord::Base.connection.

select_all(„SELECT * FROM movies ...“)
select_one(„SELECT * FROM movies ...“)
...
```

- Liefert einen Array mit Hashes oder Hashes

```
h = [{:title => „Casablanca“, :id => 1}, {...}]

h[0][:title]
```

Erinnerung

- models/student.rb

```
def fehlende_vl

  erg = []
  self.lectures.each do |l|
    erg = erg + (l.vorgaenger - self.lectures)
  end

  return erg.uniq

end
```

Erinnerung

- models/student.rb

```
def fehlende_vl
  Lecture.find_by_sql("SELECT l.* FROM lectures
  l, lectures_students ls, lectures_lectures ll
  WHERE ls.lecture_id = ll.nachfolger_id and
  ls.student_id = " + self.id.to_s + " and
  ll.vorgaenger_id = l.id and ll.vorgaenger_id
  not in (SELECT ls2.lecture_id FROM
  lectures_students ls2 WHERE ls2.student_id =
  " + self.id.to_s + ")")
end
```

Erinnerung

- Ergebnis:

```
Loaded suite test/unit/student_test
Started
..
Finished in 0.097540 seconds.

2 tests, 2 assertions, 0 failures,
0 errors, 0 skips
```

Zusammenfassung

- Datenbankabstraktion mit ORM und ActiveRecord
- Applikationsdesign mit Model-View-Controller
- Don‘t Repeat Yourself
- Convention over Configuration
- Scaffolds & Migrations
- Bestehende Datenbanken und neue Applikationen
- Tests

Literatur und weitere Informationen

- Agile Web Development with Rails.
  Thomas, Dave; Heinemeier Hansson, David.
  (Obacht: Fourth Edition für Rails 3, Rest veraltet!)
- Rails API:
  http://api.rubyonrails.org/
- Ruby-Dokumentation:
  http://corelib.rubyonrails.org/
- Rails Guides:
  http://guides.rubyonrails.org/