

# Rails Conditions

Domain Language und Business Regeln

Norman Timmler

- Was ist eine Domain Language?
- Warum haben wir für Business Regeln beim XING Marketplace einen Rails Logik Layer eingeführt?
- Wie entstand daraus das Rails Conditions Plugin?

Dave Thomas: ... If you show the code to the customer and they understand it, at least in principle, then you're programming at the right level.

# Domain Language

- eine Sprache, die einfach ist
- eine Sprache, die alle verstehen
- eine Sprache, die „systemübergreifend“ ist

English works well.

„There are only two hard things in CS:  
cache invalidation and naming things.“

— Phil Karlton via Tim Bray

# Domain Language

Projektmanager:

„User generated content steht bei uns im Mittelpunkt!“

```
class Member  
end
```

```
class User  
end
```

# Domain Language

- dieselbe Domain Language als Grundlage für den Code als auch für das Gespräch
- diese Domain Language durchgängig benutzen
- eine einheitliche Domain Language durch technische Spezifikation

# Domain Language

- Wie finde ich gute Namen für meine Modelle? ✓
- An welche Modelle knüpfe ich meine Methoden?

# Domain Language

```
class User  
end
```

```
class Posting  
end
```

User aktiviert das Posting.

# Domain Language

## Aktive Form

```
class User
  def activate_posting!(posting)
    ...
  end
end
```

+ Subjekt, Prädikat, Objekt

- Posting Logik im User Modell

# Domain Language

## Passive Form

```
class Posting
  def activate!(user)
    ...
  end
end
```

- Nicht so gut „lesbar“

+ Posting Logik im Posting Modell

# Domain Language

## Delegation

```
class Posting
  def activate!(user)...
end
```

```
class User
  def activate_posting!(posting)
    posting.activate!(self)
  end
end
```

- Der Code ist über die Modelle verstreut

Lösung: Logik Layer

# Rails Rules

# Rails Rules

## Beispiel

```
namespace :postings do
  condition :user_owns_posting do
    posting.user_id = user.id
  end

  logic :user_activates_posting,
    :pre => :user_owns_posting do
    posting.active = true
    posting.active_period = 30.days
    posting.save!
  end
end
```

# Rails Rules

## Aufruf einer Logik

```
class PostingsController
  before_filter :load_user

  def activate
    @posting = Posting.find(params[:id])
    LOGIC.ex(
      :postings, # namespace
      :user => @user,
      :posting => @posting
    ).user_activates_posting!
  end
end
```

# Rails Rules

Features

# Rails Rules

## Logiken thematisch sortieren

```
namespace :postings
  logic :user_activates_posting do
    ...
  end
end

namespace :messages
  logic :user_receives_expiration_email do
    ...
  end
end
```

# Rails Rules

## Automatischer Transaktionskontext

```
logic :user_activates_posting do
  # Transaktion Start
  posting.active = true
  posting.save!
  PostingActivation.create!(
    :user => user,
    :posting => posting
  )
  # Transaktion Ende
end
```

# Rails Rules

## Execution context

```
LOGIC.ex(:postings, :user => @user,  
  :posting => @posting) do |ec|  
  
  ec.user_activates_posting  
  ec.reset_posting_expiration_period  
  
end
```

# Rails Rules

## Conditions werfen Exceptions

```
def activate
  @posting = Posting.find(params[:id])

  LOGIC.ex(:postings,
    :user => @user, :posting => @posting
  ).user_activates_posting!

rescue UserOwnsPostingFailed
  render :template => 'shared/error'
end
```

# Rails Rules

**FAILED**

# Rails Rules

FAILED

- der Logik Aufruf war nicht DRY

```
LOGIC.ex(  
  :postings, # namespace  
  :user => @user,  
  :posting => @posting  
)  
.user_activates_posting!
```

# Rails Rules

FAILED

- die meisten Features haben wir nicht benötigt („Design auf Vorrat“)
- viel mehr Conditions als Logik

# Standard Rails

OK

```
class Posting
  class UserOwnsPostingFailed < Exception; end

  def activate!(user)
    unless user_id == user.id
      raise UserOwnsPostingFailed
    end
    update_attribute(:active, true)
  end
end
```

```
class Posting
  class UserOwnsPostingFailed < Exception; end

  def activate!(user)
    unless user_id == user.id
      raise UserOwnsPostingFailed
    end
    update_attribute(:active, true)
  end
end
```

```
class User
  class OwnsPostingFailed < Exception; end
end
```

```
class Posting
  def activate!(user)
    unless user_id == user.id
      raise User::OwnsPostingFailed
    end
    update_attribute(:active, true)
  end
end
```

```
class User
  class OwnsPostingFailed < Exception; end

  def owns_posting?(posting)
    posting.user_id == id
  end
end
```

```
class Posting
  def activate!(user)
    unless user.owns_posting?(self)
      raise User::OwnsPostingFailed
    end
    update_attribute(:active, true)
  end
end
```

```
class User
  class OwnsPostingFailed < Exception; end

  def owns_posting?(posting)
    posting.user_id == id
  end

  def owns_posting!(posting)
    raise OwnsPostingFailed unless
      owns_posting?(posting)
  end
end

class Posting
  def activate!(user)
    user.owns_posting!(self)
    update_attribute(:active, true)
  end
end
```

# Rails Conditions

COOL

```
class User
  condition :owns_posting do |posting|
    posting.user_id == id
  end
end
```

```
class Posting
  def activate!(user)
    user.owns_posting!(self)
    update_attribute(:active, true)
  end
end
```

# Rails Conditions

Vorteile gegenüber Rails Rules

- Logik und Conditions im Modell
- ein Makro und keine komplexe DSL

# Rails Conditions

Vorteile gegenüber Standard Rails

- DRY
- ein Makro generiert drei zusammenhängende Bestandteile

# Rails Conditions

## Codegenerator

```
class Posting

  condition :online do
    expires_at >= Time.now
  end

  # generierte Methoden:
  def online?... # Predicate Method
  def online!... # Bang Method

  class Error < Exception; end
```

# Rails Conditions

## Rückgabewerte

```
condition :online do  
  expires_at >= Time.now  
end
```

```
# Predicate Method
```

```
@posting.online? # -> true / false
```

```
# Bang Method
```

```
@posting.online! # -> true / OnlineFailed
```

# Rails Conditions

## Conditions ohne Block

```
class Posting
  condition :active

  # durch Rails generiert
  def active?
    self[:active]
  end

  # generierter Code:
  def active!...

  class Error < Exception; end
```

# Rails Conditions

## Preconditions

```
class Posting

  condition :online do
    expires_at >= Time.now
  end

  condition :can_be_edited, :if => :online

end
```

# Rails Conditions

## Komplexe Preconditions

```
class Posting

  condition :online do
    expires_at >= Time.now
  end

  condition :can_be_edited,
    :if => :online & :active
end
```

# Rails Conditions

## Rückgabewerte komplexer Preconditions

```
condition :can_be_edited,  
          :if => :online & :active
```

```
# Predicate Method
```

```
@posting.can_be_edited? # -> true / false
```

```
# Bang Method
```

```
@posting.can_be_edited! # -> true /  
                        # CanBeEditedFailed
```

# Rails Conditions

## Exceptions komplexer Preconditions

```
condition :can_be_edited,  
  :if => :online & :active  
  
begin  
  @posting.can_be_edited!  
rescue Posting::CanBeEditedFailed => e  
  flash[:error] = 'nicht editierbar'  
end
```

# Rails Conditions

Exception Mixins komplexer Preconditions

```
condition :can_be_edited,  
  :if => :online & :active
```

```
begin
```

```
  @posting.can_be_edited!
```

```
rescue Posting::OnlineFailed => e  
  flash[:error] = 'nicht online'
```

```
rescue Posting::ActiveFailed => e  
  flash[:error] = 'nicht aktiv'
```

```
end
```

# Rails Conditions

## Weitere komplexe Preconditions

```
condition :can_be_copied,  
  :if => :inactive | :expired
```

```
condition :can_be_deleted,  
  :if => :inactive & :offline | :expired
```

```
condition :is_showable,  
  :if => :active & :online &  
    (:user_is_admin | :user_owns_posting)
```

# Rails Conditions

Conditions anderer Modelle nutzen

```
class User
  condition :owns_posting do |posting|
    posting.user_id == id
  end
end
```

```
class Posting
  condition :can_be_deleted, :if =>
    :user_owns_posting
end
```

# Rails Conditions

Conditions anderer Modelle nutzen

```
@posting.can_be_deleted?(@user)
```

# Rails Conditions

## Preconditions ohne Conditions

```
class Posting
  condition :can_be_edited,
    :if => :online & :active
end
```

# Rails Conditions

## Conditions in Controller

```
class Posting < ActiveRecord::Base
  # Boolean field inactive
end
```

```
class User < ActiveRecord::Base
  condition :owns_posting do...

  condition :can_edit_posting, :if =>
    :user_owns_posting & :posting_inactive
end
```

# Rails Conditions

## Conditions im Controller

```
class PostingsController
  before_filter :load_user
  def edit
    @posting = Posting.find(params[:id])
    @user.can_edit_posting!(@posting)
    ...
  rescue Posting::InactiveFailed
    flash[:error] = 'Posting aktiv!'
  rescue User::OwnsPostingFailed
    redirect_to my_postings_url
  end
end
end
```

# Rails Conditions

## Conditions in der View

```
<%= link_to_if @user.can_edit_posting?(@posting),  
  'Ändern', :action => 'edit') %>
```

# Rails Conditions

at a glance

- sind „lesbar“
- lassen sich wieder verwenden (DRY)
- vereinfachen den Kontrollfluss im Controller  
(Exceptions für Ausnahmefälle statt komplexer  
if / else Strukturen)

In Planung

# Negierte Conditions

```
class Posting
  condition :can_be_activated,
    :if => :inactive

  # Generierte Methoden:
  def can_be_activated?...
  def can_be_activated!...

  # Methoden implizit durch method_missing:
  def cannot_be_activated?...
  def cannot_be_activated!...
end
```

# Download

...hoffentlich nächste Woche unter...

<http://rubyforge.org/projects/railsconditions>

## Noch Fragen?