Agile Webentwicklung mit Ruby on Rails

Prof. Dr. Oliver Vornberger
Nils Haldenwang, B.Sc.

UNIVERSITÄT OSNABRÜCK

Institut für Informatik
AG Medieninformatik

Institut für Informatik
Prof. Dr. Oliver Vornberger
Nils Haldenwang, B.Sc.

Universität Osnabrück
http://www-lehre.inf.uos.de/~ror
28.06.2012

Agile Webentwicklung mit Ruby on Rails

Sommersemester 2012

Blatt 7

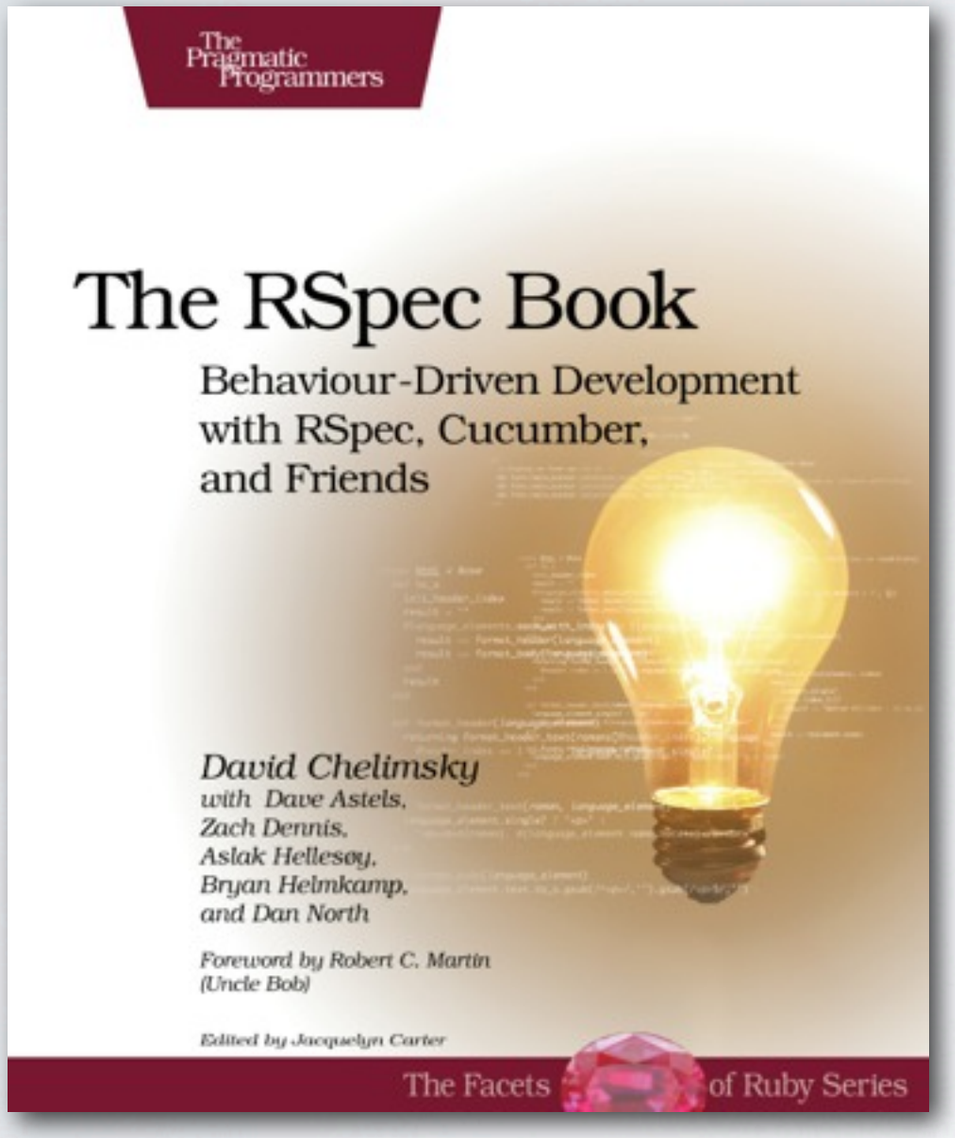
Im Ordner `blog` finden Sie die in der Vorlesung erzeugte Applikation *Blog*. Alle Aufgaben sollten nacheinander in derselben Applikation bearbeitet werden.

Aufgabe 1: Kommentare

Erweitern Sie die Applikation um eine Kommentarfunktion. Bei jedem Artikel soll ein Link angezeigt werden, über den man einen Kommentar für

Behaviour-Driven Rails

Quellen/Literatur



Chelimsky, D. und Astels, D. und Dennis, Z. und Hellesoy, A. and Helmkamp, B., *The RSpec Book*, Pragmatic Bookshelf, 2010

Einführung:
RSpec & Cucumber

RSpec

http://rspec.info/

"We use RSpec to write executable examples of the expected behavior of a small bit of code in a controlled context."

Hello RSpec

Example Group

Example

```
# rspec_greeter_spec.rb

describe "RSpec Greeter" do
  context "when greet() is called" do
    it "should say 'Hello RSpec!'" do
      greeter = RSpecGreeter.new
      greeter.greet.should == "Hello RSpec!"
    end
  end
end
```

Example ausführen

```
$ rspec rspec_greeter_spec.rb
```

```
Failures:

  1) RSpec Greeter when greet() is called should say 'Hello RSpec!'
     Failure/Error: greeter = RSpecGreeter.new
     NameError:
       uninitialized constant RSpecGreeter
     # ./rspec_greeter_spec.rb:5:in `block (3 levels) in ...

Finished in 0.00062 seconds
1 example, 1 failure

Failed examples:

rspec ./rspec_greeter_spec.rb:4 # RSpec Greeter when greet()...
```

RSpecGreeter

```
# rspec_greeter.rb

class RSpecGreeter
  def greet
    "Hello RSpec!"
  end
end
```

```
.

Finished in 0.0003 seconds
1 example, 0 failures
```

Cucumber

http://cukes.info/

"Cucumber reads plain-text descriptions of application features with example scenarios and uses the scenario steps to automate interaction with the code being developed."

Hello Cucumber

```
# features/greeter_says_hello.feature

Feature: Greeter says hello

  In order to start learning RSpec and Cucumber
  As a reader of The RSpec Book
  I want a greeter to say Hello

  Scenario: Greeter says hello
    Given a greeter
    When I send it the greet message
    Then I should see "Hello Cucumber!"
```

Feature ausführen

```
$ cucumber features
```

```
Feature: Greeter says hello

  In order to start learning RSpec and Cucumber
  As a reader of The RSpec Book
  I want a greeter to say Hello

  Scenario: Greeter says hello                 # ..._hello.feature:9
    Given a greeter                            # ..._hello.feature:10
    When I send it the greet message           # ..._hello.feature:11
    Then I should see "Hello Cucumber!" # ..._says_hello.feature:12

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.002s
```

Step Definitions

```
Given /^a greeter$/ do
  @greeter = CucumberGreeter.new
end

When /^I send it the greet message$/ do
  @message = @greeter.greet
end

Then /^I should see "(.*?)"$/ do |greeting|
  @message.should == greeting
end
```

Feature ausführen

```
[...]

  Scenario: Greeter says hello                 # ..._hello.feature:9
    Given a greeter                            # step_definitions/greeter_steps.rb:1
      uninitialized constant CucumberGreeter (NameError)
      ./step_definitions/greeter_steps.rb:2:in `/^a greeter$/'
      features/greeter_says_hello.feature:10:in `Given a greeter'
    When I send it the greet message           # step_definitions/greeter_steps.rb:5
    Then I should see "Hello Cucumber!" # step_definitions/greeter_steps.rb:9

Failing Scenarios:
cucumber features/greeter_says_hello.feature:9 # Scenario: Greeter says hello

1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
0m0.002s
```

Steps nach einem fehlschlagenden Step werden übersprungen

Cucumber erkennt, wo die Step-Defintion sich befindet

CucumberGreeter

```
class CucumberGreeter
  def greet
    "Hello Cucumber!"
  end
end
```

```
[...]

1 scenario (1 passed)
3 steps (3 passed)
0m0.002s
```

Zusammenspiel von RSpec und Cucumber

Anwendungsgebiete

RSpec:

Spezifikation von Objekten

Cucumber:

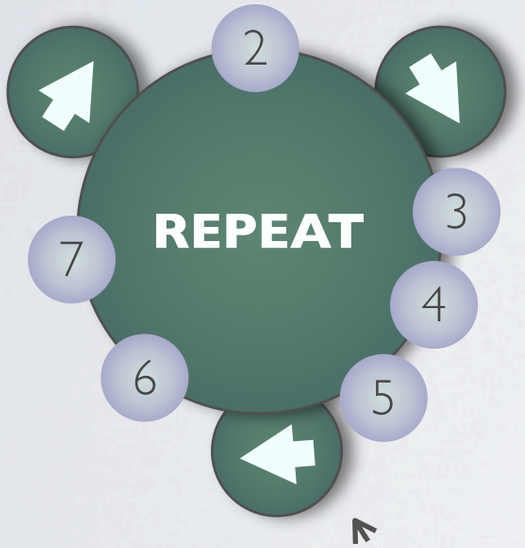
Spezifikation der Systemfunktionalität

BDD Cycle

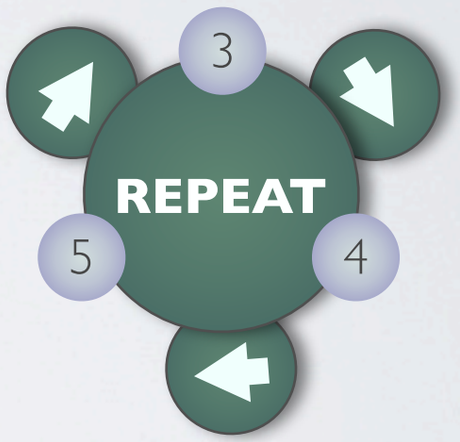
When Scenario is passing

1. Focus on one scenario
2. Write failing step definition

Drop down to RSpec



3. Write failing example
4. Get the example to pass
5. Refactor



6. When step is passing

7. Refactor

Simulation der Anwenderinteraktion

Direct Model Access

```
Given /^a movie$/ do
   @movie = Movie.create!
end
```

- Schnelle Ausführung
- Nicht aus Anwendersicht
- Umgeht Controller und View

Simulated Browser

- Kein echter Browser
- JavaScript nicht ausführbar
- Geschicktes Parsen und Verarbeiten des HTML-Codes
- Interaktion durch Aufruf von Links und Absenden von Formularen

Simulated Browser

Webrat:

https://github.com/brynary/webrat/

Capybara:

https://github.com/jnicklas/capybara/

Simulated Browser: Webrat

```
When /^I create ‘Caddyshack’ in the Comedy genre $/ do
  visit movies_path
  click_link "Add Movie"
  fill_in "Title", with: "Caddyshack"
  select "1980", from: "Release Year"
  check "Comedy"
  click_button "Save"
end
```

- Testet das Zusammenspiel aller Schichten aus Anwendersicht
- Größerer Aufwand führt zu längeren Laufzeiten

Automated Browser

Startet im Hintergrund einen Browser ohne Oberfläche und steuert diesen automatisiert

Selenium:

http://seleniumhq.org/

selenium-client:

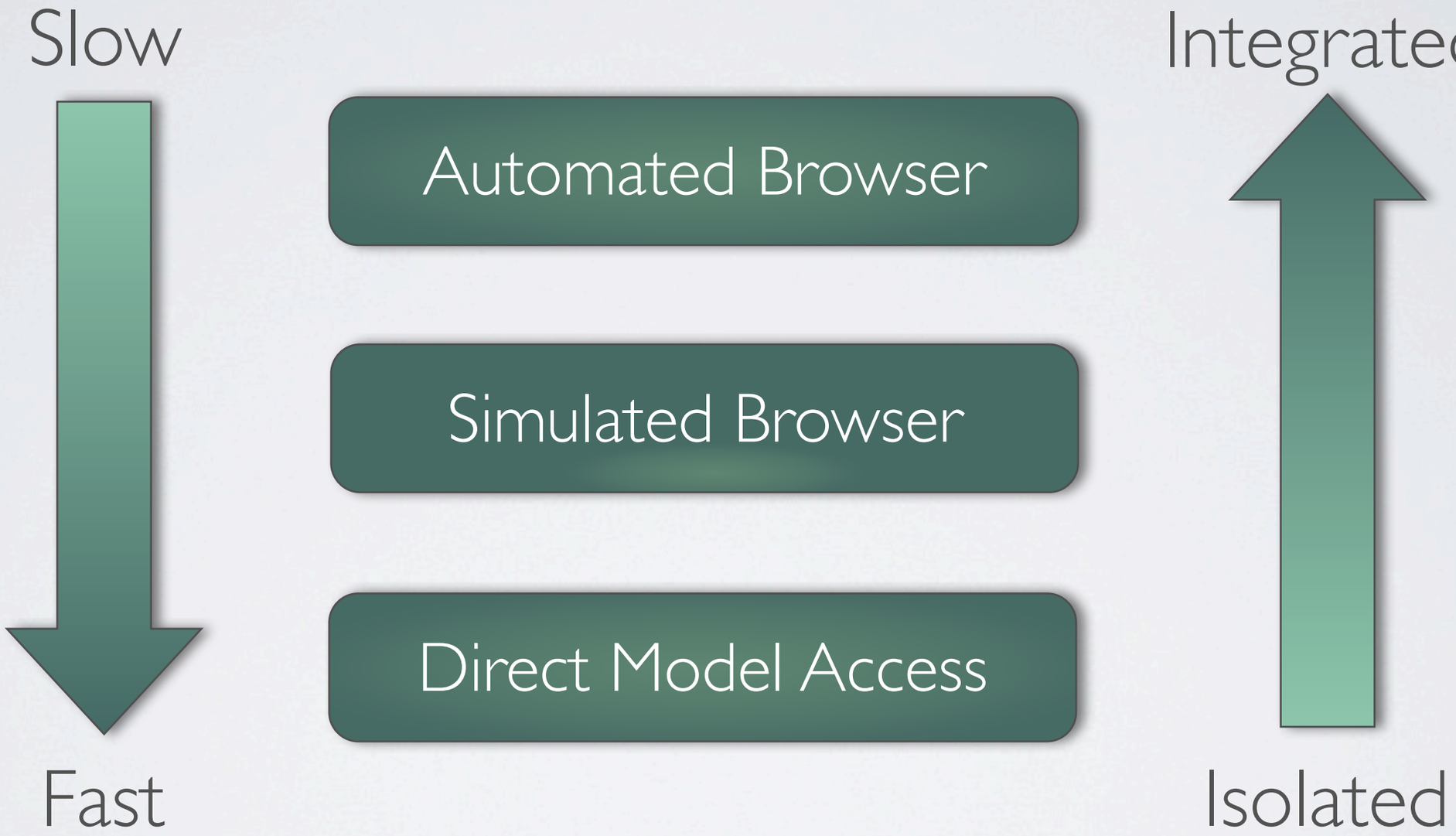
https://github.com/ph7/selenium-client/

Automated Browser

API ist kompatibel mit Webrat/Capybara

- Testet Zusammenspiel aller Schichten aus Anwendersicht
- Ermöglicht Testen von JavaScript
- Größerer Aufwand führt zu längeren Laufzeiten
- Aufsetzen und Debugging der Testumgebung ist sehr komplex

Übersicht


Slow
Integrated
Automated Browser
Simulated Browser
Direct Model Access
Fast
Isolated

Richtlinien

- Simulated Browser für *When* und *Then*
- Direct Model Access für *Given* (sofern die eigentliche Funktionalität bereits getestet ist)
- Simulated Browser für *Given*, wenn Browserstatus hergestellt werden muss (z.B. Login-Session)
- Automated Browser für kritische JavaScript- und AJAX-Features (nur *happy path*)

View Specs

View Specs

Idee:

Die Oberfläche ist der zentrale Punkt der Nutzerinteraktion. Daraus wird deutlich, welche Models, Controller und Methoden zur Erfüllung der Anforderungen nötig sind.

Vorgehen:

Stelle die Daten bereit und prüfe, ob sie korrekt dargestellt werden.

View Specs: Beispiel

View rendern

Test-Double eines ActiveRecord-Objektes

```
describe "messages/show.html.erb" do
  it "displays the text attribute of the message" do
    message = double("Message", text: "Hello World!")
    assign(:message, message)
    render
    rendered.should contain("Hello World!")
  end
end
```

Ergebnis des Renderns

Zuweisung einer Variablen

Mocking Models

```
message = mock_model("Message").as_new_record.as_null_object
```

Simuliert ungespeichertes Objekt

Ignoriert unbekannte Methodenaufrufe

- Erlaubt Spezifikation der View, obwohl noch kein Model existiert
- Vermeidet die Erzeugung eines Datensatzes in der Datenbank

Wann sollte man View-Specs einsetzen?

Am I using Cucumber and Webrat?

Views-Specs sind oft nicht nötig, da sie nur Dinge spezifizieren, die bereits spezifiziert sind

Will a Cucumber failure give me the right message?

Bei komplexen Dingen können View-Specs aufschlussreicher sein

Wann sollte man View-Specs einsetzen?

Is there any functionality beyond basic CRUD actions/views?

View-Specs oft nicht notwendig bei Standardoberflächen, spezifiziere nur davon abweichende, komplexe Views

Controller Specs

Controller Specs

Rails' controllers are like waiters in a restaurant. A customer orders a steak dinner from a waiter. The waiter takes the request and tells the kitchen that he needs a steak dinner. When the steak dinner is ready, the waiter delivers it to the customer for her enjoyment.

- Craig Demyanovich

Idee:

Wenn sich der Controller nicht in Isolation testen lässt, dann tut er mehr, als er sollte

Controller Specs

Vorgehen:

Simuliere einen Request und prüfe die zugewiesenen Instanzvariablen

Controller Specs müssen keine Views rendern

Komplexe Modeloperationen können über Mocks und Stubs simuliert werden

Beispiel

```
describe MessagesController do
  describe "POST create" do
    it "creates a new message"
    it "saves the message"
  end
end
```

Example Groups können geschachtelt werden

Beispiel

```
it "creates a new message" do
  Message.should_receive(:new).with(text: "Text")
  post :create, message: { text: "Text" }
end
```

Fehler beheben: Model und Route anlegen

```
def create
end
```

Fehler: Template messages/ create.html.erb fehlt

Example pausieren

Verhindert Ausführung

```
it "creates a new message" do
  pending("drive out redirect")
  Message.should_receive(:new).with(text: "Text")
  post :create, message: { text: "Text" }
end

it "redirects to the Messages index" do
  post :create
  response.should redirect_to(action: :index)
end
```

Enthält Antwortobjekt mit Request-Informationen

Redirect implementieren

```
def create
  redirect_to action: :index
end
```

Nur so viel implementieren, wie das Example verlangt!

Neues Objekt anlegen

```
it "creates a new message" do
  Message.should_receive(:new).with(text: "Text")
  post :create, message: { text: "Text" }
end
```

```
def create
  Message.new(params[:message])
  redirect_to action: :index
end
```

Example läuft durch

Objekt speichern

```
it "saves the message" do
  message = mock_model(Message)
  Message.stub(:new).and_return(message)
  message.should_receive(:save)
  post :create
end
```

```
def create
  message = Message.new(params[:message])
  message.save
  redirect_to action: :index
end
```

Example “saves the message” läuft durch

Fail: “creates new message”

```
it "creates a new message" do
  Message.should_receive(:new).with(text: "Text")
  post :create, message: { text: "Text" }
end
```

Überschreibt alte new-Methode, Controller erwartet aber Rückgabewert, auf dem *save* aufgerufen wird

“creates new message”

```
it "creates a new message" do
  message = mock_model(Message)

  Message.should_receive(:new).
    with(text: "Text").
    and_return(message)

  post :create, message:
         { text: "Text" }
end
```

Rückgabewert für Message-Expectation

Schlägt noch fehl: Der Aufruf von *save* wird von *message* nicht erwartet

Lösung:

```
message = mock_model(Message).as_null_object
```

Refactored Examples

```
describe MessagesController do
  describe 'POST create' do
    let(:message) { mock_model(Message).as_null_object }

    before(:each) do
      Message.stub(:new).and_return(message)
    end

    it "creates a new message"                # ...
    it "saves the message"                    # ...
    it "redirects to the Messages index" # ...
  end
end
```

Refactored Examples

```
it "creates a new message" do
  Message.should_receive(:new).
    with(text: "Text").and_return(message)
  post :create, message: { text: "Text" }
end

it "saves the message" do
  message.should_receive(:save)
  post :create
end

it "redirects to the Messages index" do
  post :create
  response.should redirect_to(action: :index)
end
```

Model Specs

Model Specs

If Rails controllers are like waiters in a restaurant, Rails models are the kitchen staff. They know how to cook a steak to order.

\- Zach Dennis

Cucumber Steps, View- und Controller-Specs haben offenbart, welche Modelklassen notwendig sind. Diese beeinhalten die Geschäftslogik und sollten in Isolation vom Rest der Applikation getestet werden.

Beispiel

```
describe Message do
  it "is valid with valid attributes"

  it "is not valid without a title"

  it "is not valid without text"
end
```

```
class Message < ActiveRecord::Base
end
```

3 examples, 0 failures, 3 pending

“is valid with valid attributes”

Funktioniert für alle Methoden mit ?
am Ende des Namens

```
it "is valid with valid attributes" do
  Message.new.should be_valid
  # <=> Message.new.valid?.should == true
end
```

3 examples, 0 failures, 2 pending

“is not valid without a title”

```
it "is not valid without a title" do
  message = Message.new(title: nil)
  message.should_not be_valid
end
```

```
class Message < ActiveRecord::Base
  validates_presence_of :title
end
```

3 examples, 1 failures, 1 pending

Fail: "is valid with valid attributes"

```
it "is valid with valid attributes" do
  Message.new(title: "Title").should be_valid
end
```

3 examples, 0 failures, 1 pending

“is not valid without text”

```
it "is not valid without text" do
  message = Message.new(text: nil)
  message.should_not be_valid
end
```

3 examples, 0 failures, 0 pending

False Positive

Test läuft durch, weil *message* wegen fehlendem *title* invalid ist

“is not valid without text”

```
it "is not valid without text" do
  message = Message.new(text: nil, title: “Title”)
  message.should_not be_valid
end
```

```
class Message < ActiveRecord::Base
  validates_presence_of :title, :text
end
```

“is valid with valid attributes” muss noch angepasst werden

3 examples, 0 failures, 0 pending

Refactored Examples

```
describe Message do
  before(:each) do
    @message = Message.new(title: "Foo", text: "Bar")
  end

  it "is valid with valid attributes" do
    @message.should be_valid
  end

  it "is not valid without a title" do
    @message.title = nil
    @message.should_not be_valid
  end

  it "is not valid without text" # analog ...
end
```

Erzeugen von Testdaten

Fixtures

```
# test/fixtures/users.yaml
willi:
  first_name: Willi
  last_name: Wacker
  admin: false
```

```
it "should return its frist name" do
  users(:willi).first_name.should == "Willi"
end
```

- Definition der Daten weit verstreut und nicht direkt in der Spec ersichtlich

Factory Girl

https://github.com/thoughtbot/factory_girl

Default-Werte

Admin erbt Werte von User, kann sie aber überschreiben

```
FactoryGirl.define do
  factory :user do
    first_name "first_name"
    last_name "last_name"
    admin false

    factory :admin do
      admin true
    end
  end
end
```

Verwendung

```
# Returns a User instance that's not saved
user = FactoryGirl.build(:user)

# Returns a saved User instance
user = FactoryGirl.create(:user)

# Returns a hash of attributes that can
# be used to build a User instance
attrs = FactoryGirl.attributes_for(:user)

# Build a User instance and override the
# first_name property
user = FactoryGirl.build(:user, first_name: "Joe")
user.first_name # => "Joe"
```

Zusammenfassung

- Anforderungen vom Stakeholder erfassen und in Cucumber-Features übersetzen
- Für jedes Scenario jedes im Rahmen einer Iteration entwickelten Features den BDD-Cycle durchführen
- Mit der View beginnen, daraus ergeben sich nötige Controller und daraus wiederum die Models mit ihren Methoden
- Durch regelmäßiges Refactoring und enge Kooperation mit den Stakeholder entsteht eine den Anforderungen entsprechende Applikation, mit einem qualitativ hochwertigen Softwaredesign