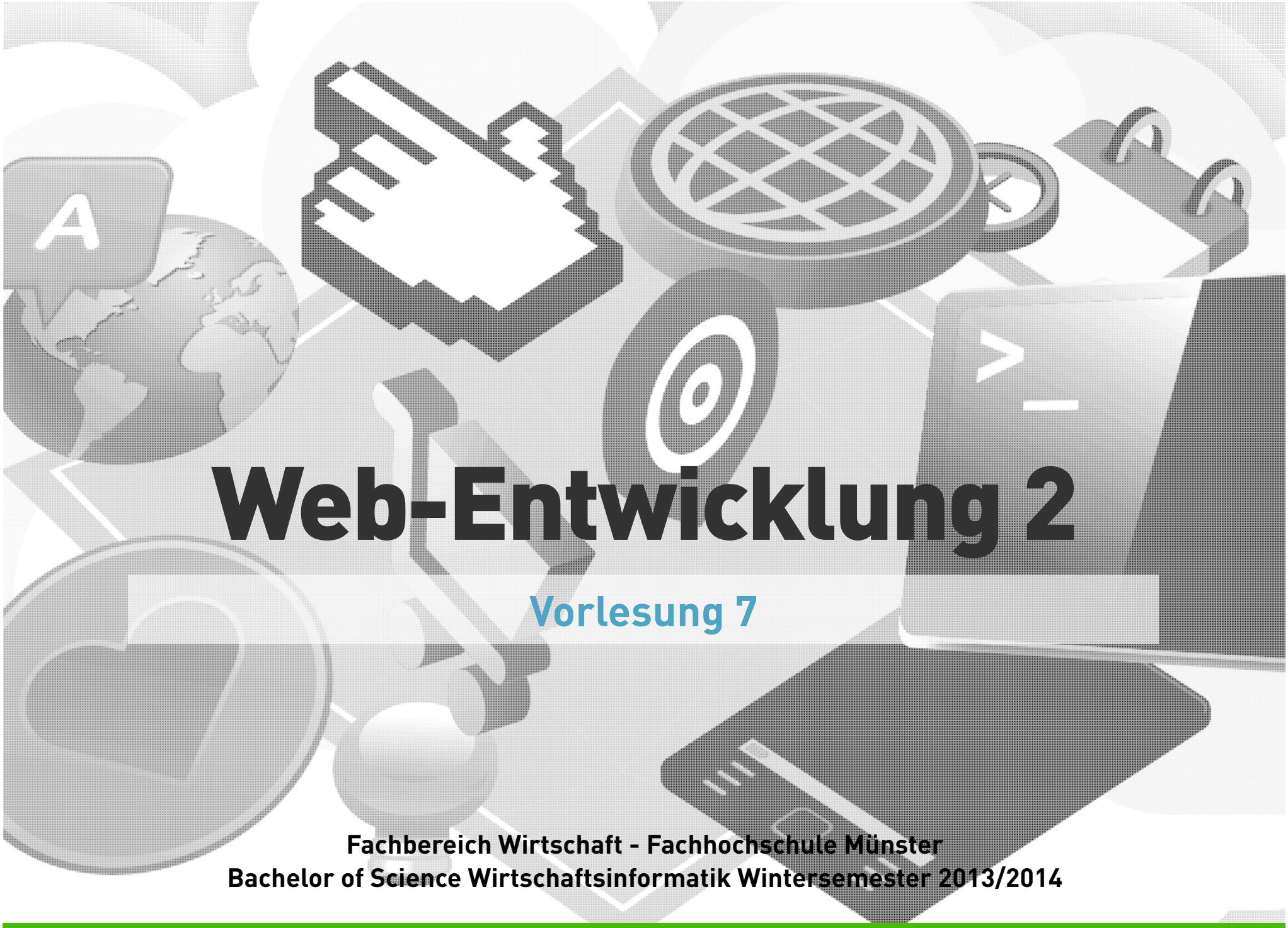


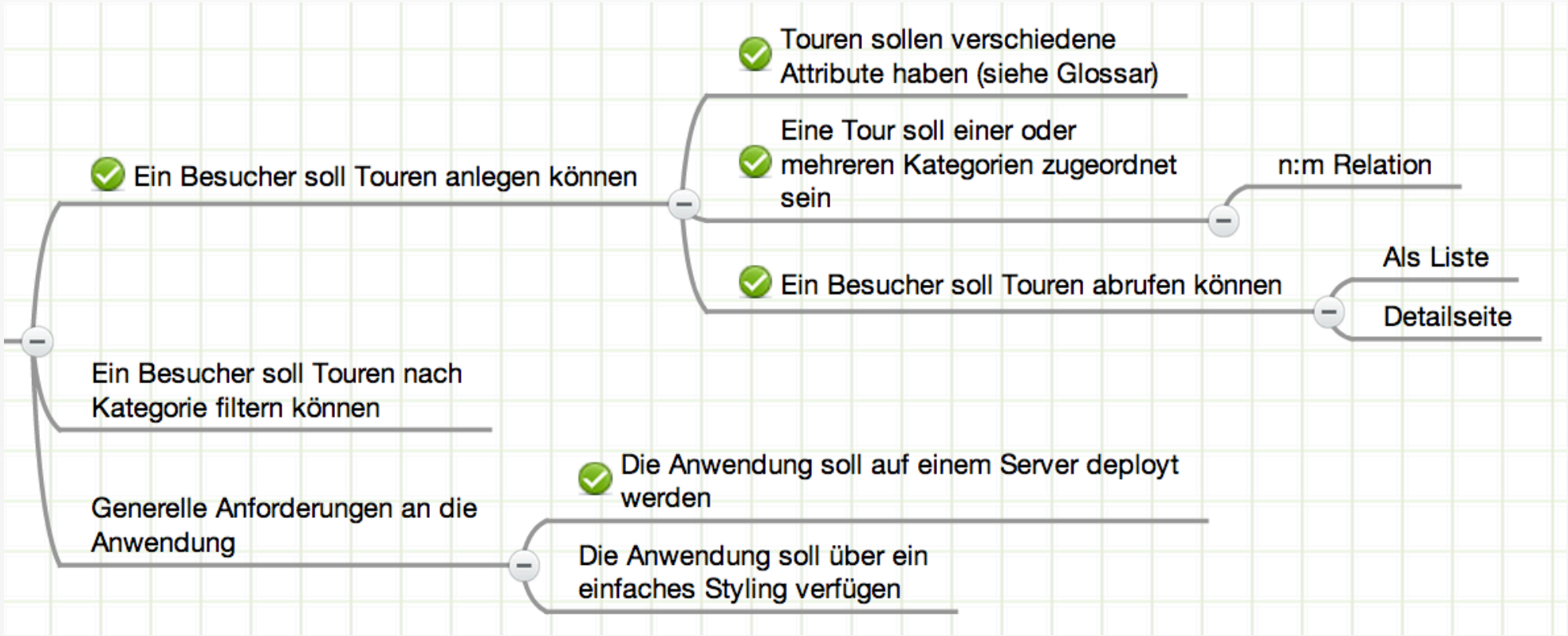
Web-Entwicklung 2

Vorlesung 7

Fachbereich Wirtschaft - Fachhochschule Münster
Bachelor of Science Wirtschaftsinformatik Wintersemester 2013/2014

Was war?

RECAP



Taskboard

Inbox	In Progress	In Review	Complete
Ein Anwendung soll über ein einfaches Styling verfügen		Ein Besucher soll Touren anlegen können, ...	
Ein Besucher soll Touren nach Kategorie filtern können, ...	Ein Besucher soll Touren nach Kategorie filtern können, ...		

Exkurs: Testgetriebene Entwicklung

Was ist ein Test?

EXKURS

Softwaretest (i.w.S.):
Softwaretest ist ein Test während der Softwareentwicklung, um die Funktionalität einer Software an den Anforderungen und ihre Qualität zu messen, und Softwarefehler zu ermitteln

Softwaretest (i.e.S.):
Softwaretest ist ein Programm, welches Software auf bestimmte Kriterien überprüft.

Warum TDD?

- Wasserfallmodell: Testphase erst nach Umsetzungsphase
- Probleme:
  - Testabdeckung oft mangelhaft
  - Entwickelte Komponenten sind schwierig zu testen
  - Entwicklung neuer Features führt oft zu Regressionsfehlern
- Testgetriebene Entwicklung: Gemeinsame Entwicklung von Tests und Features

Testarten

- Unit-Test
  - Test einer einzelnen Einheit / Klasse / Modul
  - Laufen isoliert voneinander ab
- Functional Test
  - Zusammenhängende Tests mehrerer Module / Klassen
- Integrationstest
  - Überprüfung des fehlerfreien Zusammenwirkens von Systemkomponenten
- Penetrationstest
  - Test des gesamten Systems unter Extrem-Bedingungen
- Akzeptanztest
  - Test durch Auftraggeber/Anwender

Testverfahren

EXKURS

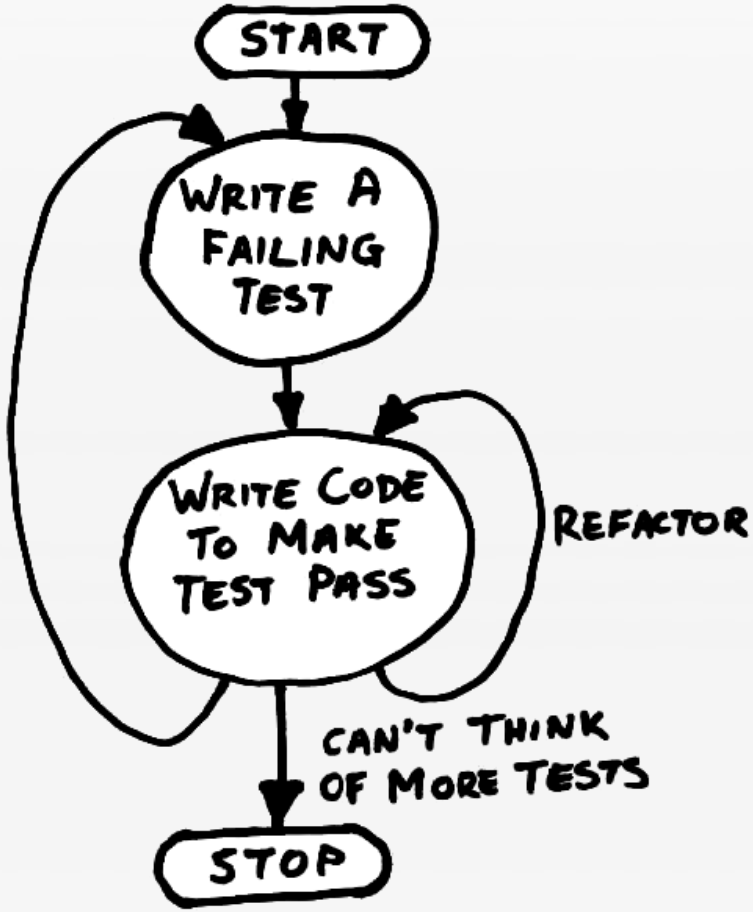
- Manuelle Prüfverfahren:
  - Programminspektion, Review, Walkthrough
  - Zeit- und Personalaufwendig
- Blackbox-Test (funktionaler Test)
  - Test ohne Kenntnis der internen Funktionsweise des Systems
  - Sicherstellung der gewünschten Funktionalität
- Whitebox-Test (struktureller Test)
  - Test mit Kenntnis der internen Funktionsweise des Systems
  - Häufig unter Anwendung von Metriken

TDD-Zyklus

Red - Green - Refactor

TDD-Zyklus

EXKURS



Vorteile von TDD

EXKURS

- Hohe Testabdeckung wird erzielt, weil kein Feature ohne entsprechenden Test entwickelt wird
- Regressionsfehler werden vermieden
- Einfacheres Refactoring von Code möglich
- Entwickelte Komponenten sind i.d.R. stärker entkoppelt
- Tests "dokumentieren" die Anwendung

Nachteile von TDD

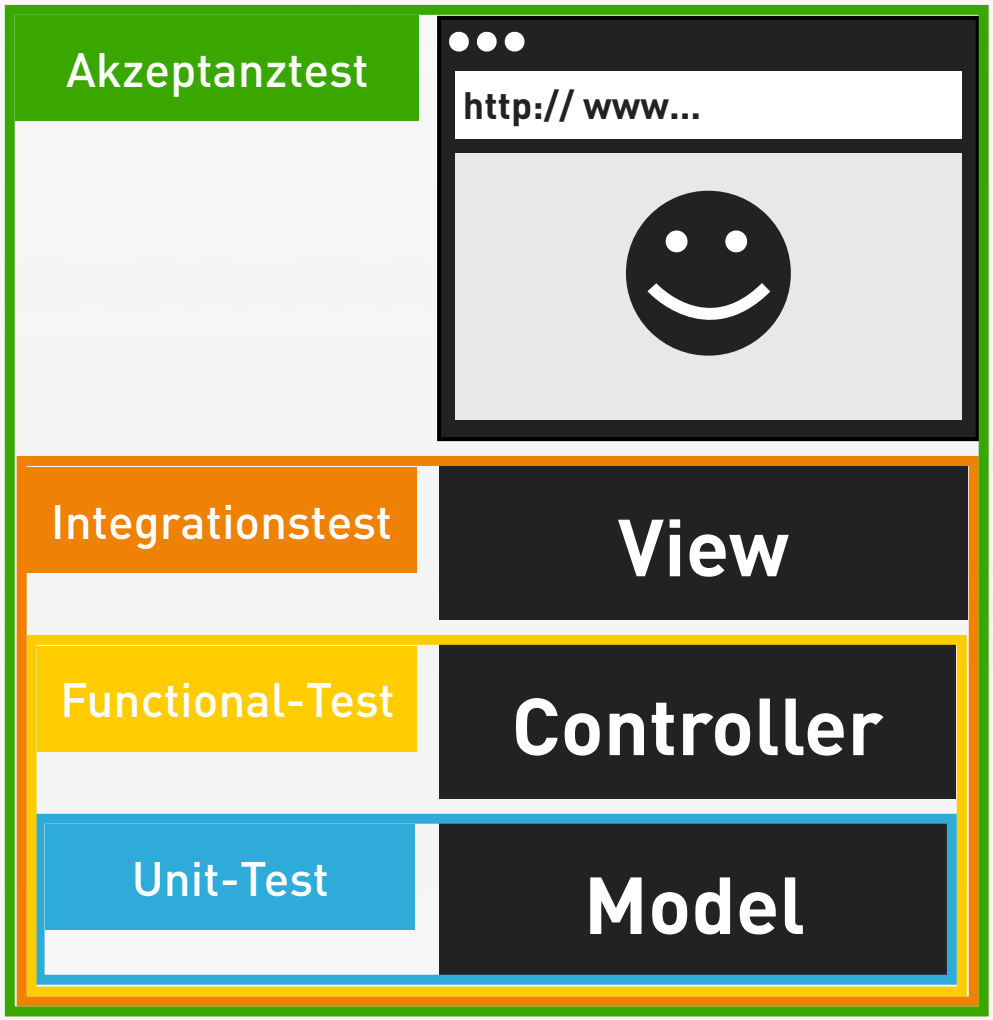
EXKURS

- Initial höher Zeitaufwand
- Schreiben von Tests vor Features erfordert Umdenken
- Durchlaufende Testsuite garantiert keine hundertprozentige Fehlerfreiheit

ABER:
Der Aufwand lohnt sich langfristig IMMER!

Test einer RoR-Anwendung

EXKURS



Test einer RoR-Anwendung

- Erfolgreiche testgetriebene Entwicklung benötigt Toolunterstützung:
  - Tests müssen einfach zu schreiben sein
  - Tests müssen leicht verständlich sein
  - Tests müssen automatisiert laufen

Kritischer Faktor:
Wir benötigen geeignete Test-Frameworks!

Testframeworks für Ruby on Rails

- Unit-Tests:
  - Test::Unit
  - RSpec
- Functional Tests:
  - Test::Unit
  - RSpec
- Integrationstests:
  - Test::Unit + Capybara
  - RSpec + Capybara
- Akzeptanztests:
  - Cucumber + Capybara

RSpec

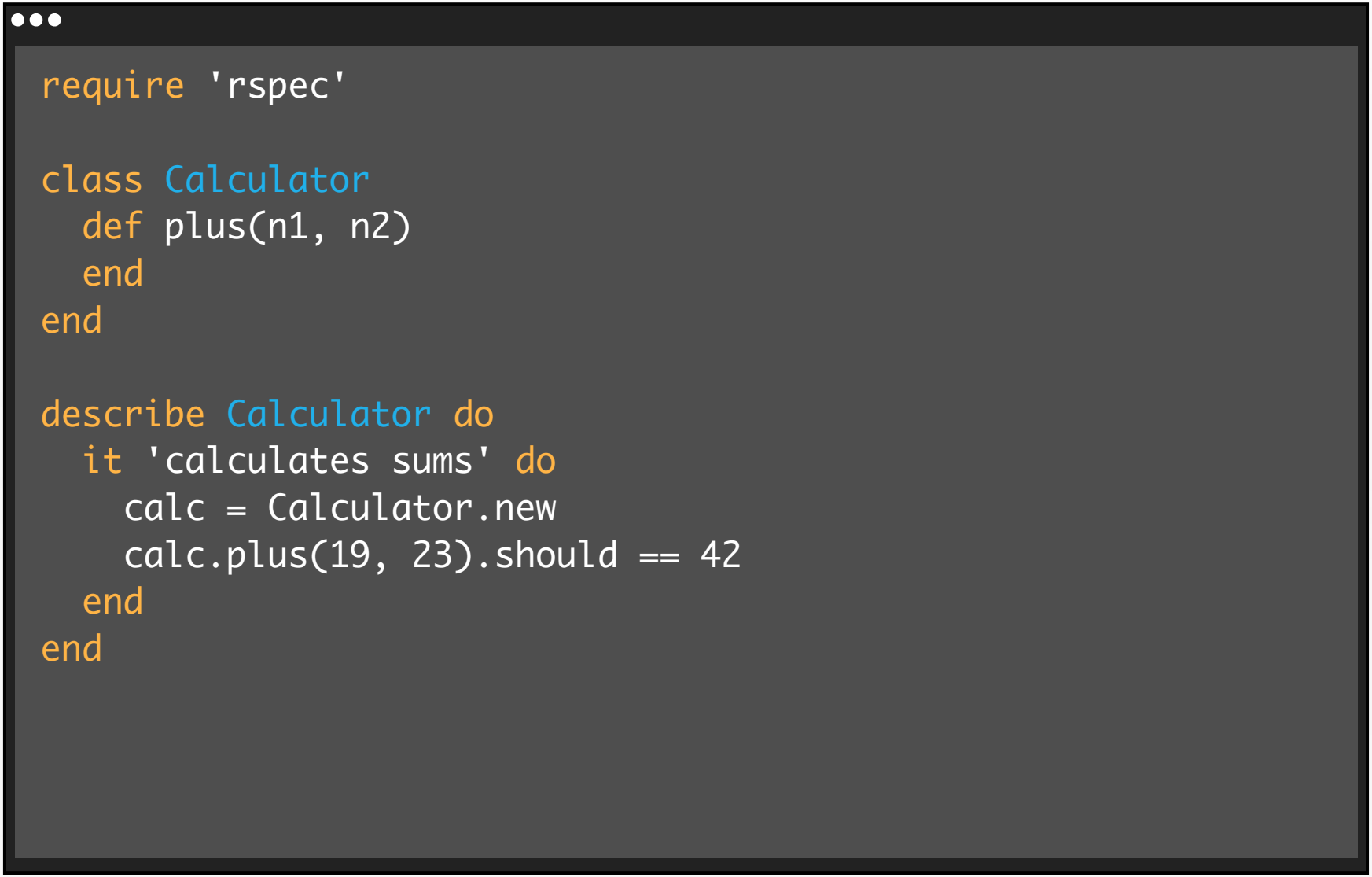
- Testing-Framework für Ruby
- Unterstützt Behaviour-Driven Development (BDD)
- Tests werden in einer DSL ("Domain-specific language") spezifiziert

- Grundlegende Elemente:
  - Expectations
  - Mocks

RSpec - Expectations

- Expectations sind Erwartungen, die ich an meinen Code stelle
- Beispiel: Ich erwarte, dass meine Funktion einen bestimmten Wert zurückliefert
- Syntax:
  actual.should == expected
- Test läuft durch, wenn alle Erwartungen zutreffen

RSpec - Expectations



```ruby
require 'rspec'

class Calculator
  def plus(n1, n2)
  end
end

describe Calculator do
  it 'calculates sums' do
    calc = Calculator.new
    calc.plus(19, 23).should == 42
  end
end
```

RSpec - Expectations

EXKURS

```
thomas@t420: ~/Coding/Ruby
thomas@t420 ~/Coding/Ruby % rspec calculator_spec.rb

Calculator
  calculates sums (FAILED - 1)

Failures:

  1) Calculator calculates sums
     Failure/Error: calc.plus(19, 23).should == 42
       expected: 42
            got: nil (using ==)
     # ./calculator_spec.rb:11:in `block (2 levels) in <top (required)>'

Finished in 0.00038 seconds
1 example, 1 failure

Failed examples:

rspec ./calculator_spec.rb:9 # Calculator calculates sums
thomas@t420 ~/Coding/Ruby %                                          :(
```

RSpec - Expectations

EXKURS

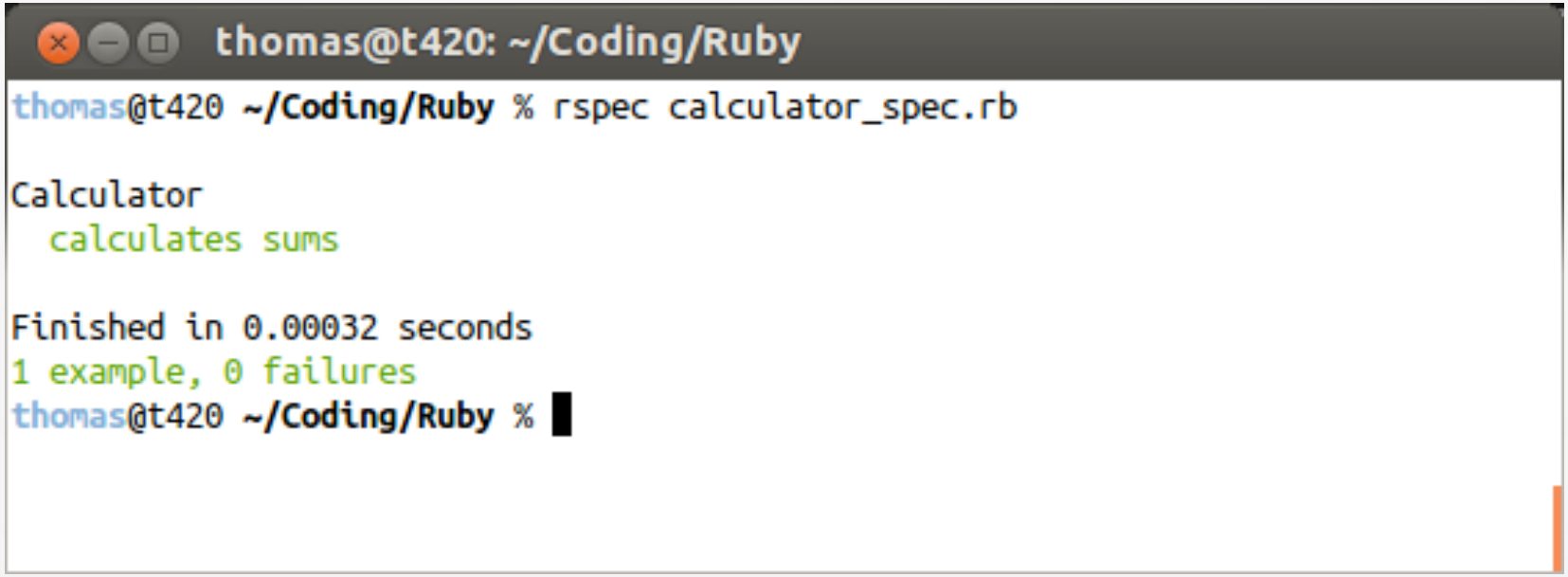
```
require 'rspec'

class Calculator
  def plus(n1, n2)
    n1 + n2
  end
end

describe Calculator do
  it 'calculates sums' do
    calc = Calculator.new
    calc.plus(19, 23).should == 42
  end
end
```

RSpec - Expectations

EXKURS



RSpec - Mocks

- Mocks verändern das Verhalten von Methoden für Testzwecke
- Beispiel: Rückgabewert einer Methode festlegen
- Syntax:
  - object.stub(:method).and_return(value)
  - object.should_receive(:method).and_return(value)

RSpec - Mocks

EXKURS

```
require 'rspec'

class Calculator
  def plus(n1, n2)
    n1 + n2
  end
end

describe Calculator do
  it 'calculates sums' do
    calc = Calculator.new
    calc.stub(:plus).and_return(42)
    calc.plus(19, 23).should == 42
  end
end
```

RSpec meets Rails

EXKURS

- RSpec lässt sich leicht in Rails integrieren:
- Gem rspec-rails ins Gemfile aufnehmen:
  gem 'rspec-rails'
  $ bundle install
- Install-Generator ausführen:
  $ rails generate rspec:install
- Ausführen aller Specs:
  $ rake spec

RSpec meets Rails

EXKURS

- rspec-rails erzeugt für generierte Klassen automatisch Testcode
- Specs befinden sich im Ordner spec/
- Konfiguration der Testsuite befindet sich in spec/spec_helper.rb

```
thomas@t420: ~/blog
thomas@t420 ~/blog (git)-[master] % rails generate model Category name:string
      invoke  active_record
      create    db/migrate/20121125102335_create_categories.rb
      create    app/models/category.rb
      invoke    rspec
      create      spec/models/category_spec.rb
thomas@t420 ~/blog (git)-[master] %
```

RSpec meets Rails

EXKURS

```ruby
require 'spec_helper'

describe Category do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

Weitere TDD-Tools

EXKURS

- factory_girl
  Tool zur einfachen Erstellung von Testdaten
- capybara
  DSL zur Fernsteuerung eines Browsers
- guard
  Führt Tests nach Änderungen automatisch aus

Weitere TDD-Tools

- database_cleaner
  Räumt die Datenbank nach Tests auf
- rcov / simplecov
  Berechnet die Testabdeckung
- faker
- ...

User Story 2

Ein Besucher soll Touren nach Kategorie filtern können, um schneller die passende Tour zu finden.

Unit-Test:
Eine Tour muss einer Kategorie zugeordnet werden.

User Story 2

- Test anlegen

spec/models/tour_spec.rb

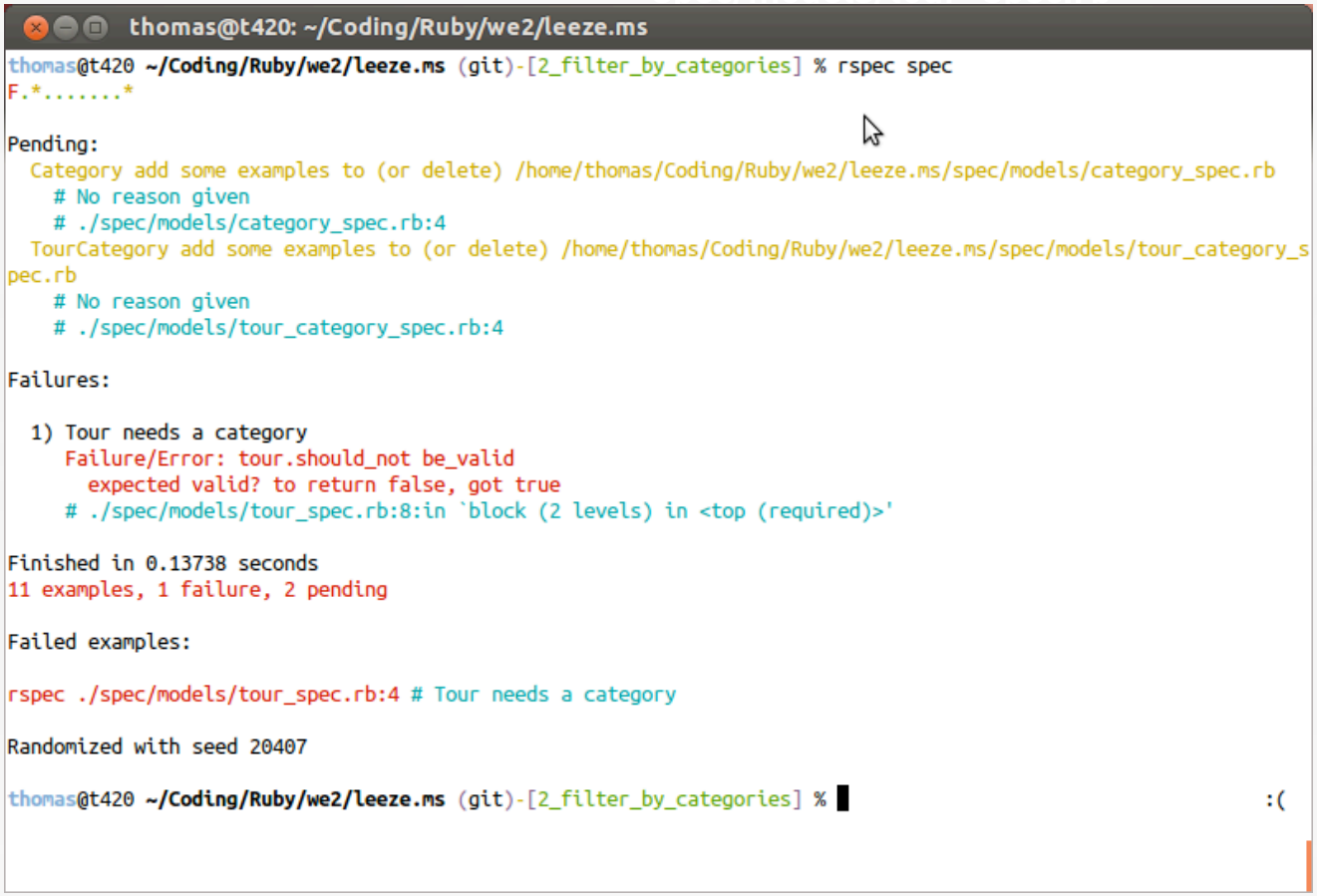
```
require 'rspec'

describe Tour do
  it 'needs a category' do
    category = Category.new name: 'Radwandern'
    tour = Tour.new title: 'Test Tour', teaser: 'My Teaser',
description: 'My description', url: 'http://www.google.com/
maps/'

    tour.should_not be_valid
    tour.categories << category
    tour.should be_valid
  end
end
```

User Story 2

- Test ausführen



User Story 2

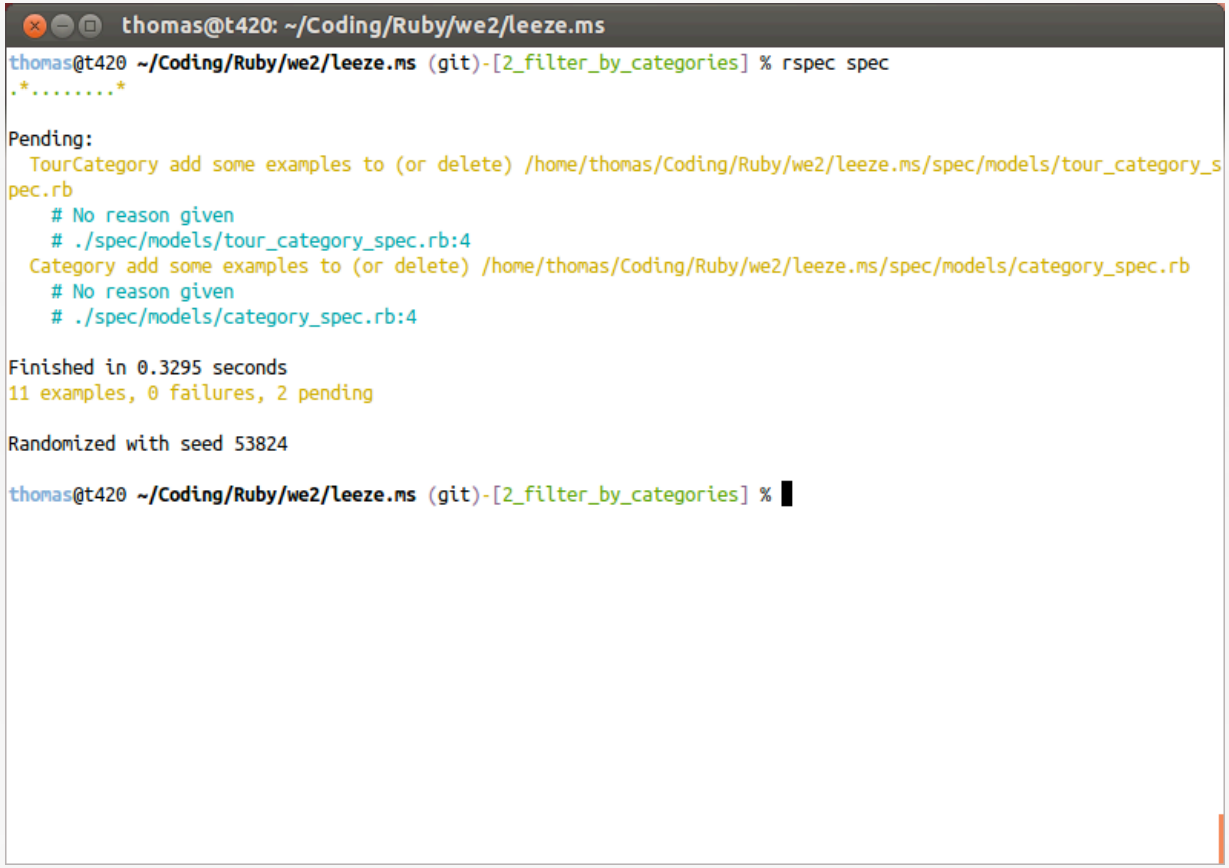
- Code schreiben

app/models/tour.rb

```
class Tour < ActiveRecord::Base
  has_many :tour_categories
  has_many :categories, through: :tour_categories
  validates :title, :teaser, :description, :url, presence: true
  validates :url, format: %r|\Ahttp(s?)://www.google.com/maps/|
  validates :categories, presence: true
end
```

User Story 2

- Test ausführen



User Story 2

- Test refactorn

spec/models/tour_spec.rb

```
require 'rspec'

describe Tour do
  let!(:category) { Category.new name: 'Radwandern' }
  let!(:tour) { Tour.new title: 'Test Tour', teaser: 'My Teaser', description: 'My
description', url: 'http://www.google.com/maps/' }

  it 'is not valid without a category' do
    tour.should_not be_valid
  end

  it 'is valid with a category' do
    tour.categories << category
    tour.should be_valid
  end
end
```

User Story 2

tbc.