HPI Hasso Plattner Institut

Behavior-Driven (Software) Development

Software Engineering II
WS 2020/21

Enterprise Platform and Integration Concepts

Image by *Jay* on flickr: https://www.flickr.com/photos/activars/6590201161/ (CC BY-NC 2.0)

Agenda

1. Why Behavior-driven Development (BDD)?
   - **Goals of Automated Testing**
   - Writing Software that Matters
2. Basics of Tests and BDD
3. Advanced Concepts & Testing Tests
4. Outlook

Automated Testing Use Case

Feature 1

Manual testing	TDD/BDD, little manual testing
Minute 5: working registration page **Minute 8: feature is tested (3 times)**	Minute 05.00: working test Minute 10.00: working implementation **Minute 10.30: feature is tested (3 times)**

Assumptions: 1min manual testing, 10s automatic test

Automated Testing Use Case

Feature 2: Special case for feature 1

Manual testing	TDD/BDD, little manual testing
Minute 11: implemented Minute 14: tested (3 times)	Minute 12.30: test ready **Minute 15.30: implemented** Minute 16.00: tested (3 times)

Automated Testing Use Case

Feature 2: Special case for feature 1

Manual testing	TDD/BDD, little manual testing
Minute 11: implemented	Minute 12.30: test ready
Minute 14: tested (3 times)	Minute 15.30: implemented
	Minute 16.00: tested (3 times)
Minute 17: refactoring ready	***Minute 19.00: refactoring ready***
Minute 19: tested feature 1	Minute 19.10: tested both features
Minute 21: tested feature 2	**Minute 20.10: committed**
Minute 22: committed	

Automated Testing

Goals

- Find errors **faster**
- Better code (correct, robust, maintainable)
- Less overhead when testing -> tests are used **more frequently**
- Easier to modify existing features, **refactoring**

But

- Tests might have **bugs**
- Test environment != production environment *(what could help here?)*
- Tests **must be maintained** (and refactored)

Agenda

1. Why Behavior-driven Development (BDD)?
   - Goals of Automated Testing
   - **Writing Software that Matters**
2. Basics of Tests and BDD
3. Advanced Concepts & Testing Tests
4. Outlook

Writing Software that Matters

HPI

"BDD is about implementing an application by describing its behavior from the perspective of its stakeholders"

– Dan North

Principles

- Unified language between business and technology
- Systems should have identified, verifiable stakeholder value
- Up-front analysis, design and planning have diminishing returns

Related: YAGNI

https://behaviourdriven.org/

BDD Implementation

Specify behavior in unified way, e.g.

- As a User Story (semi-formal language)
  - **Narrative:** As a <role>, I want <feature>, so that <value>
  - **Acceptance criteria:** Given <context>, When <event>, Then <outcome>
  - Can be mapped to code
- As an (automated) end-to-end test
  - **Formalized language (code)**
  - Go to URL, select …, type …, press 'OK'
  - RSpec & Capybara follow this approach

We use *'acceptance test'* and *'end-to-end'* test as synonyms

BDD Cycle

1

red 2

3

red

Unit Tests

refactor

green 4

5

refactor

green 6

7

Acceptance Tests

Adapted from [Chelimsky et al.: The Rspec Book, 2010]

Definition of Done

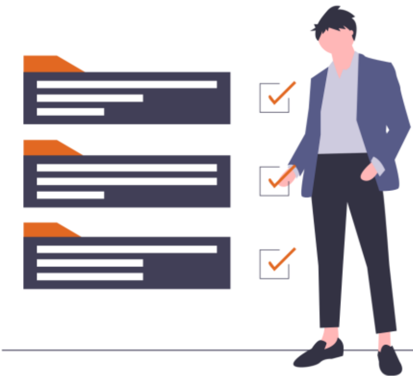
How do I know when to stop? E.g.

- Acceptance criteria fulfilled
- All tests are green

But also (?)

- Objective quality standards are met
- Second opinions
- Secure
- Documented

Definition of Done:
A team's consensus of what it takes to complete a feature.



BDD Cycle

HPI

Hierarchy of tests/requirements

- Tests should be specified in terms of desired behavior
- **Focus on Big Picture** (vs. implementation detail)
  - Big picture: end-to-end test,
    e.g. visit website, modify data
  - Technical implementation: unit test,
    e.g. this method returns only ints
- BDD can be considered **"outside-in"**
  - From coarse to fine
  - Acceptance/end-to-end tests outside
  - Unit/other tests inside

Of course, technical implementation is still required

Maximum BDD Pyramid

- Vision
- Goals
- Epics
- Use Cases | Features
- User Stories | Scenarios
- Scenario Steps
- Test Cases

Vision

All Stakeholders, one statement

- *Example:* Improve Supply Chain

Core stakeholders define the vision

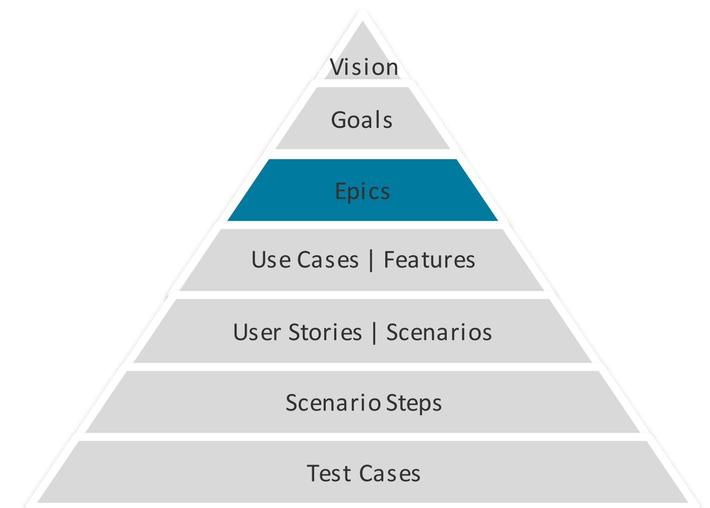
- Incidental stakeholders help understand
  - What is possible
  - At what cost
  - With what likelihood

Goals

- How the vision will be achieved.
- Examples
  - Easier ordering process
  - Better access to suppliers' information

Epics

- Huge themes / feature sets are described as an "epic"
- Too high level to start coding but useful for conversations
- Examples
  - Reporting
  - Customer registration

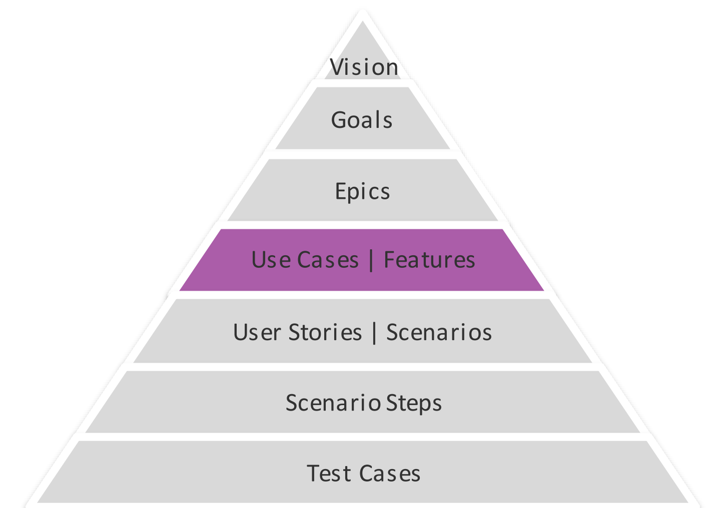


Use Cases / Features

- Describe the behavior we will implement in software
- Can be traced back to a stakeholder

- ***Warning:*** Do not directly start at this level

- Explain the value & context of a feature to stakeholders
  - Not too much detail
- Features deliver value to stakeholders

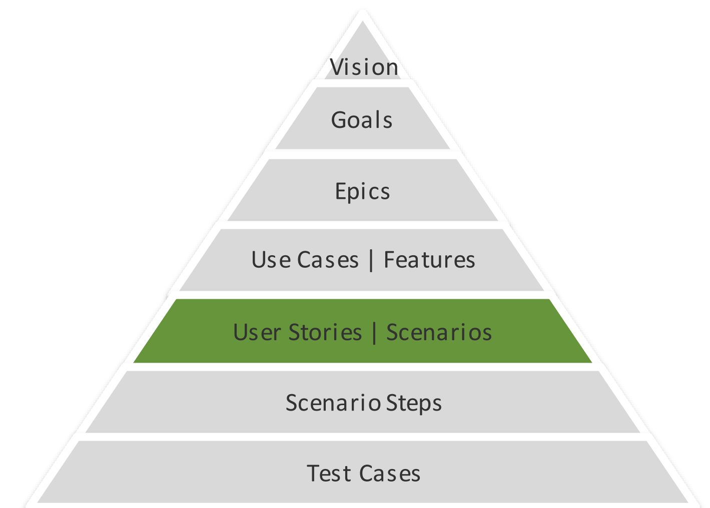


User Stories

User Stories are demonstrable functionality

- 1 Feature -> 1..n User Stories
- Stories should be vertical (e.g. no database-only stories)
- User stories are tokens for conversations
- Attributes (**INVEST**)
  - **I**ndependent
  - **N**egotiable
  - **V**aluable (from a business Point of View)
  - **E**stimable
  - **S**mall enough to be implemented in one iteration
  - **T**estable

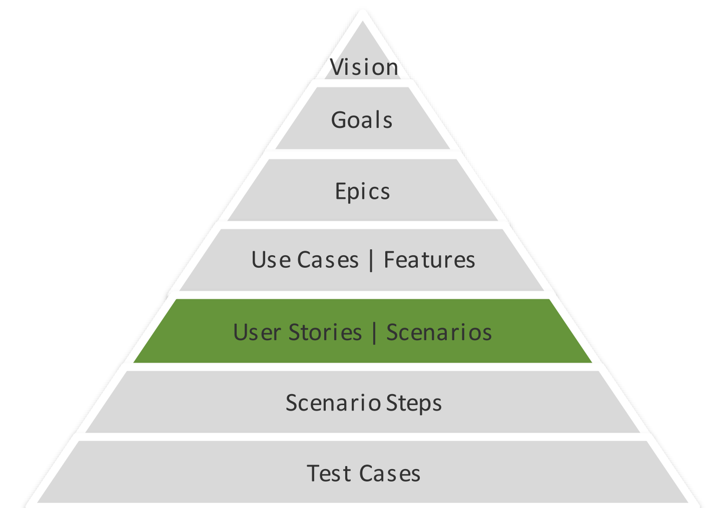
See http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/



User Stories

Story content

- Title
- Narrative
  - **Description, reason, benefit** *(why?)*
  - “As a <stakeholder>, I want <feature> so that <benefit>”
  - “In order to <benefit>, a <stakeholder> wants to <feature>”
- Acceptance criteria
  - Criteria for what needs to be implemented for PO to accept story
  - Related to Definition of Done



Scenarios, Steps, Test Cases

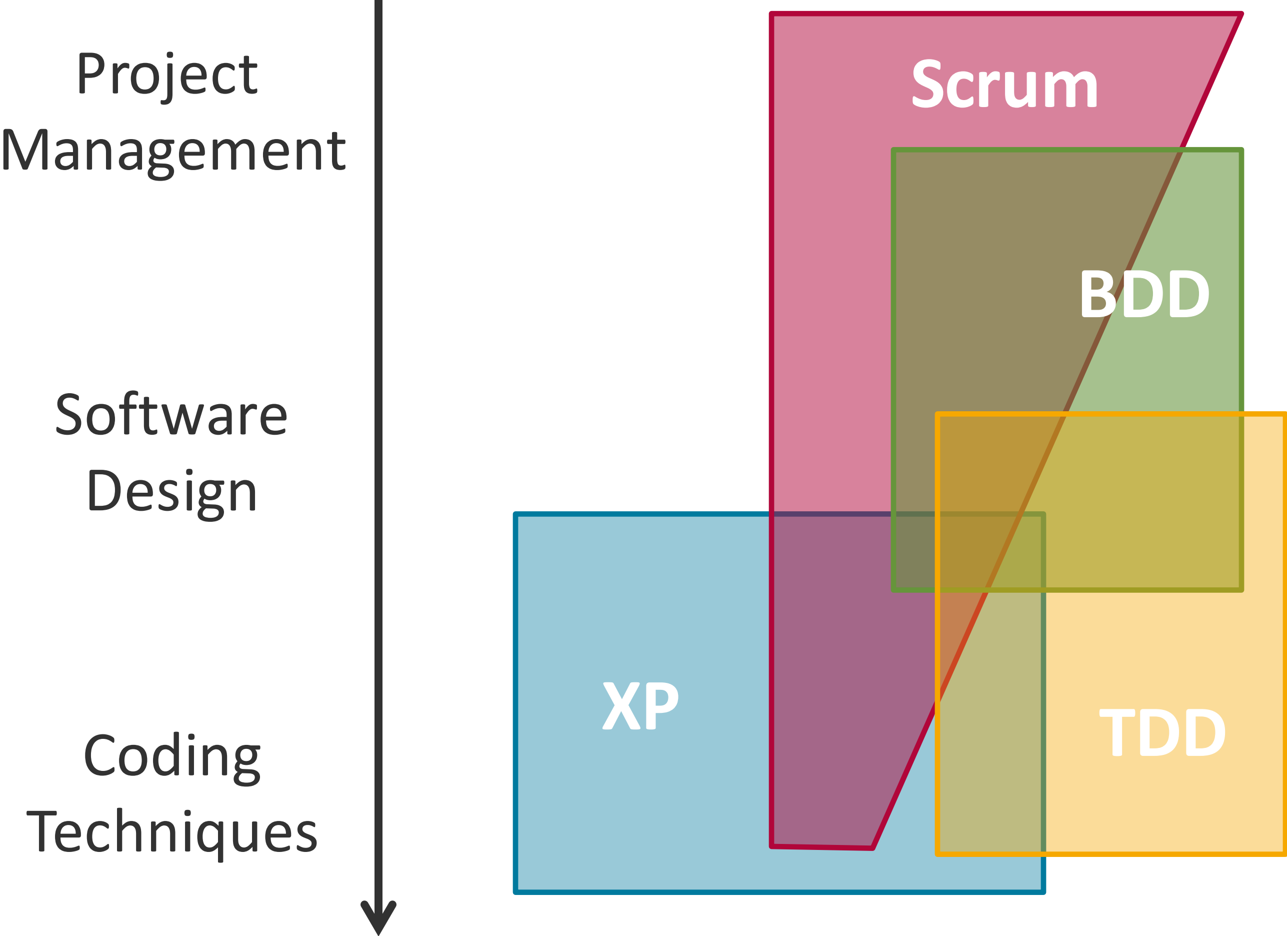
Scenarios

- 1 User Story -> 1..n scenarios
- Each scenario describes one aspect of a User Story
- Describe high-level behavior

Scenario steps

- 1 scenario -> m scenario steps + step implementation
- 1 scenario step -> 0..i tests
- Describe low-level behavior

Agile Methods & BDD



Behavior-driven Development

For stakeholders

- Story-based definition of application behavior
- Definition of features to reach goal & vision
- Business value is specified in requirements

For the developer

- BDD Cycle, definition of stories/tests with PO
- Coding with TDD/test-first approach
- Automated testing

Summary

Behavior-driven Development

- Motivation
- Automated testing vs manual
- Levels of abstraction
  - Vision -> test cases

BDD Cycle

- Definition of Done
- Pros & Cons

Agile & BDD

- Where does BDD fit
- BDD vs TDD

HPI Hasso Plattner Institut

Introduction to Testing in Ruby on Rails

Software Engineering II
WS 2020/21

Enterprise Platform and Integration Concepts

Image by *wiredforlego* on flickr: https://www.flickr.com/photos/wiredforsound23/6862675420/ (CC BY-SA 2.0)

Agenda

1. Why Behavior-driven Design (BDD)?
2. **Basics of Tests and BDD**
   - Model Tests
   - View Tests
   - Controller Tests
   - Integration & Acceptance Tests
3. Advanced Concepts & Testing Tests
4. Outlook

Test::Unit vs. RSpec

- Test::Unit comes with Ruby

```
class UserTest < Test::Unit::TestCase

  def test_first_name
    user = User.new
    assert_nil user.name, "User's name was not nil."
    user.name = "Chuck Norris"
    assert_equal user.first_name, "Chuck", "user.first_name did not return 'Chuck'."
  end

end
```

RSpec: Rails Testing Framework

HPI

- **RSpec offers syntactical sugar** over Rails default (Test::Unit)
- Many built-in modules
- `rspec` command with tools to constrain what examples are run

All following code refers to **RSpec 3.2**

```
describe User do
  it "should determine first name from name" do
    user = User.new
    expect(user.name).to be_nil
    user.name = "Chuck Norris"
    expect(user.first_name).to eq "Chuck"
  end
end
```

http://blog.thefirehoseproject.com/posts/test-driven-development-rspec-vs-test-unit/

RSpec Structure

Using `describe` and `it` like in a conversation

- "Describe an order!" "It sums prices of items."
- `describe` creates a test group
- `it` declares tests within group
- `context` for nested groups / structuring

- Aliases
  - Declare groups using `describe` or `context`
  - Declare individual tests using `it` or `example`

```
describe Order do
  context "with one item" do
    it "sums prices of items" do
      # ...
    end
  end

  context "with no items" do
    it "shows a warning" do
      # ...
    end
  end
end
```

https://github.com/rspec/rspec-core/blob/master/README.md

RSpec Matchers

HPI

General structure of RSpec expectation (assertion):

- expect(…).to <matcher>, expect(…).not_to <matcher>

```ruby
# Object identity
expect(actual).to be(expected) # passes if actual.equal?(expected)
# Object equivalence
expect(actual).to eq(expected) # passes if actual == expected
# Comparisons
expect(actual).to be >= expected
expect(actual).to be_between(minimum, maximum).inclusive
expect(actual).to match(/expression/) # regular expression
expect(actual).to start_with expected
# Collections
expect([]).to be_empty
expect(actual).to include(expected)
```

Specialized matchers, e.g.:
`expect(actual).to respond_to(expected)`

https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers

Agenda

1. Why Behavior-driven Design (BDD)?
2. Basics of Tests and BDD
   - **Model Tests**
   - View Tests
   - Controller Tests
   - Integration & Acceptance Tests
3. Advanced Concepts & Testing Tests

Model Tests

A Rails model

- Accesses data through an **Object-relational mapping** (ORM) tool
  - Object-oriented programming languages deal with "objects"
  - Relational databases deal with scalar values (*int, string*) in tables
  - ORM translates between these worlds
- Implements **business logic**
- Is "weighty", i.e. contains most code and application logic

Model tests in Rails

- Easiest tests to write
- Test most of application logic

Model Tests

Model test hints

- Should cover **almost all of the model code**
- Do not test framework functionality like “*belongs_to*”
- Test your validations
- How many tests? Let tests drive the code -> perfect fit (test-first approach)

Minimal model test set

- One test for the **“happy-path case”** (the usual, normal way)
- One test for each code branch
- Corner cases (nil, wrong values, …), if appropriate

- **Keep each test small!** *(why?)*

Model Tests: Example

app/models/contact.rb

```ruby
class Contact < ActiveRecord::Base
  validates :name, presence: true

  def self.by_letter(letter)
    where("name LIKE ?", "#{letter}%").order(:name)
  end
end
```

spec/models/contact_spec.rb

```ruby
require 'rails_helper'

describe Contact, type: :model do

  before :each do #do this before each test
    @john= Contact.create(name: 'John')
    @tim = Contact.create(name: 'Tim')
    @jerry = Contact.create(name: 'Jerry')
  end

  #the actual test cases
  context "with matching letters" do
    it "returns a sorted array of results that match" do
      expect(Contact.by_letter("J")).to eq [@john, @jerry]
    end

    it "omits results that do not match" do
      expect(Contact.by_letter("J")).not_to include @tim
    end
  end

end
```

Agenda

1. Why Behavior-driven Design (BDD)?
2. Basics of Tests and BDD
   - Model Tests
   - **View Tests**
   - Controller Tests
   - Integration & Acceptance Tests
3. Advanced Concepts & Testing Tests

View Tests

HPI

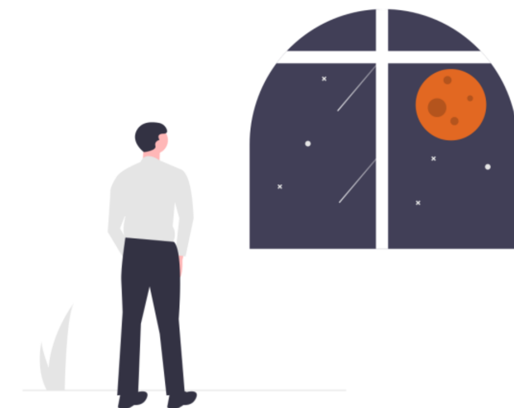
A Ruby on Rails view

- Has only minimal logic
- Should not call the database *(why?)*
- **Renders data** passed by the controller as HTML

Different terminologies:
view (RoR) ~ *template* (Django)
controller (RoR) ~ *view* (Django)

Challenges of view tests *(ideas?)*

- Time-intensive to write, HTML structure complex
- How to test look & feel?
- Brittle regarding interface redesigns



View Tests

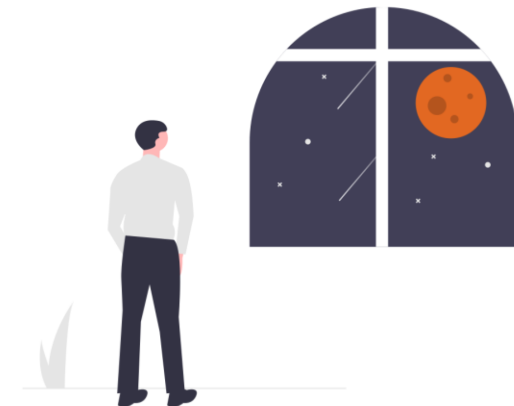
Verify the logical and semantic structure of rendered content

Goals

- Validate that view layer runs without error
- Render view templates in isolation
- Check that passed data is presented as expected
- Validate conditional display of information, e.g. based on user's role

Possible anti-patterns

- Validating HTML markup
- Checking the "design"
- Testing text that's likely to change instead of core structure elements



View Tests: Example

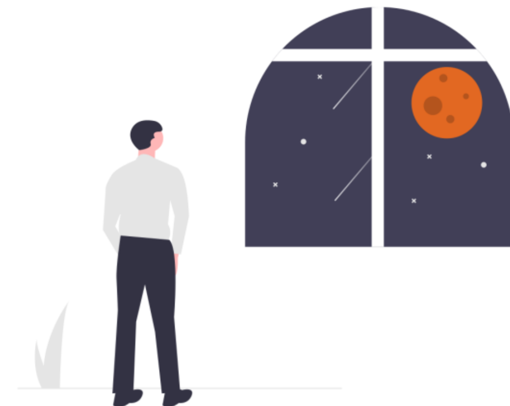
HPI

```
describe "users/index", type: :view do
  it "displays user name" do
    assign(:user,
        User.create! Name: "Bob"
    )

    # path could be inferred from test file
    render template: "users/index.html.erb"

    expect(rendered).to match /Hello Bob/
  end
end
```

`user.save!` (with !) raises an `ActiveRecord::RecordInvalid` error, when `user.save` returns false



https://www.relishapp.com/rspec/rspec-rails/v/3-2/docs/view-specs/view-spec

View Tests: Example (Capybara)

HPI

```ruby
Rspec.describe "users/index" do
  it "displays user name" do
    assign(:user,
        User.create! :name => "Bob"
    )

    # path could be inferred from test file
    render :template => "users/index.html.erb"

    # same as before
    expect(rendered).to have_content('Hello Bob')
    # a better idea
    expect(rendered).to have_css('a#welcome')
    expect(rendered).to have_xpath('//table/tr')
  end
end
```

Capybara offers many helpful "matchers", including `has_button`, `has_table` & `has_unchecked_field`

- https://github.com/jnicklas/capybara
- rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Matchers

Agenda

1. Why Behavior-driven Design (BDD)?
2. Basics of Tests and BDD
   - Model Tests
   - View Tests
   - **Controller Tests**
   - Integration & Acceptance Tests
3. Advanced Concepts & Testing Tests

Controller Tests

A Rails controller

- **Is "skinny"**, i.e. contains little code and little logic
- Retrieves appropriate data models from the database
- Calls model methods (if needed)
- Passes resulting data to the view

Goal of controller tests

- Simulate a HTTP request
- Test multiple paths through controller code, e.g. for authentication
- Verify result and the correct handling of parameters

Controller Tests

What to test in a controller test?

- Verify that requests by a user lead to...
  - Model / ORM calls
  - Correct data being handed to view
- Handling of **invalid user requests**, e.g. through redirects
- Handling of **exceptions** raised by model calls
- Verify security roles / **role-based access control**

Remember: Model functionality is tested in model tests!

Controller Tests: Ruby on Rails

RSpec controller test helpers

- Automatically imported variables
  - `request`
  - `response`
- Getter and setters for
  - Session: `session[:key]`
  - Controller variables: `assigns[:key]`
  - Flash: `flash[:key]`
- Simulate a single HTTP request
  - *get, post, put, delete*

Controller Tests: Example

```ruby
require "rails_helper"

describe TeamsController, :type => :controller do
  describe "GET index" do
    it "assigns @teams in the controller" do
      team = Team.create
      get :index
      expect(assigns(:teams)).to eq([team])
    end

    it "renders the index template" do
      get :index
      expect(response).to render_template("index")
    end
  end
end
```

http://www.relishapp.com/rspec/rspec-rails/v/3-2/docs/controller-specs

Agenda

1. Why Behavior-driven Design (BDD)?
2. Basics of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - **Integration & Acceptance Tests**
3. Advanced Concepts & Testing Tests

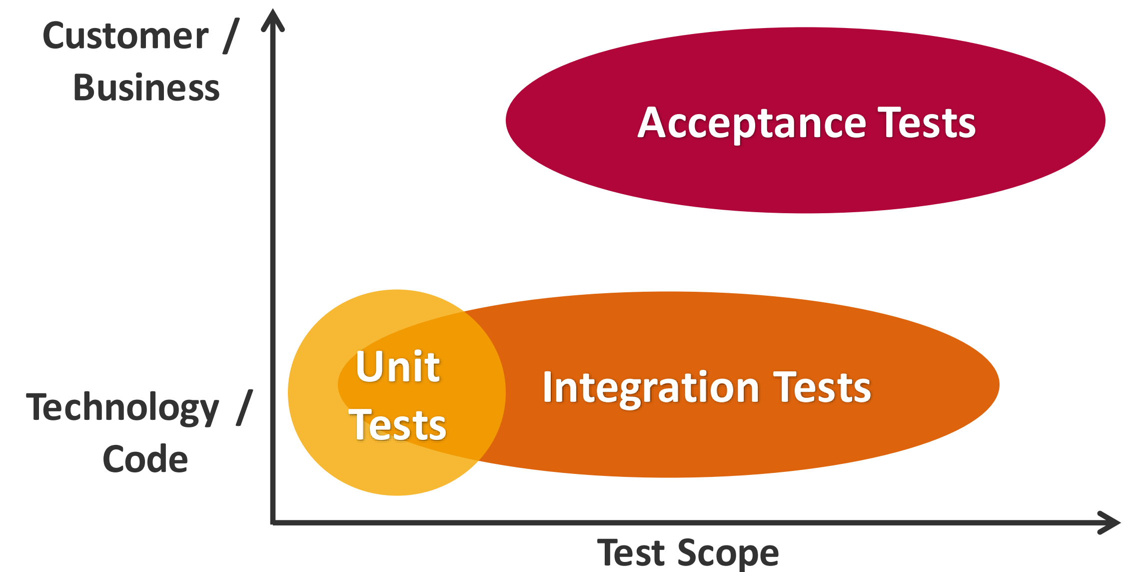
Levels of Testing

Level	Question	Automation
Staging Tests	Can the program be deployed?	Partially automatable
Quality Tests	Does the program meet quality standards?	Partially automatable
Requirement Tests (User Acceptance Tests)	Do the requirements meet the users‘ needs?	Not automatable!
Functional Tests (User Story Acceptance Tests)	Does the program functionality meet the requirements?	automatable
Integration Tests	Do the parts of the program function together?	automatable
Unit Tests	Does the code unit function?	automatable

Integration & Acceptance Tests

- Perform tests on the full system, across multiple components
- Test end-to-end functionality

- **Integration Tests**
  - Build on unit tests, **written for developers**
  - Test component interactions
- **Acceptance Tests**
  - Check if functionality satisfies the specification from a **user perspective**
  - Possibly accessible for stakeholders



- http://www.testfeed.co.uk/integration-vs-acceptance-tests/

BDD vs Test Levels

Use Cases | Features

User Stories | Scenarios

Scenario Steps

Test Cases

Requirement Tests

Functional Tests

Integration Tests

Unit Tests

BDD Implementations

Behavior-driven development (BDD)

- Story-based definition of application behavior
- Definition of features driven by business value

Implementations on different abstraction levels:

- Domain-specific languages (e.g. Cucumber)
  - Pro: Readable by non-technicians, reads like formal English
  - Cons: Extra layer of abstraction, translation to executable Ruby code
- Executable Code (e.g. using testing frameworks, RSpec & Capybara)
  - Pro: No translation overhead
  - Con: Harder to read for non-developers

Capybara Test Framework

- Simulate how a real user would interact with a web application
- Well suited for writing **acceptance & integration tests** for web applications
- Provides DSL for "surfing web pages"
  - e.g. `visit`, `fill_in`, `click_button`
- Integrates with RSpec
- Supports different "drivers", some support JavaScript evaluation
  - Webkit browser engine
  - Selenium
    - Opens an actual browser window and performs actions within it

- https://github.com/teamcapybara/capybara#using-capybara-with-rspec

Acceptance Tests (Capybara)

HPI

```ruby
require 'capybara/rspec'

describe "the signin process", :type => :feature do
  before :each do
    User.create!(:email => 'user@example.com', :password => 'password')
  end

  it "signs me in" do
    visit new_session_path
    within("#session") do
      fill_in 'Email', :with => 'user@example.com'
      fill_in 'Password', :with => 'password'
    end
    click_button 'Sign in'
    expect(page).to have_css('div#success')
  end
end
```

- *What are some issues with this test?*
- *What is good?*

Capybara includes aliases for RSpec syntax:
`feature` is `'describe …, :type => :feature'`,
`scenario` is **it, background** is **before**

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

Summary

Test Details

- RSpec Structure
- Model Tests
- View Tests
- Controller Tests
- Integration & Acceptance Tests

Testing Concepts

- Levels of testing
- When to use which type of test
- BDD implementations