Advanced Testing Concepts (in Ruby on Rails)

Scalable Software Engineering
WS 2021/22

Enterprise Platform and Integration Concepts

Agenda

Advanced Concepts & Testing Tests

- **Setup and Teardown**
- Test Data
- Test Doubles

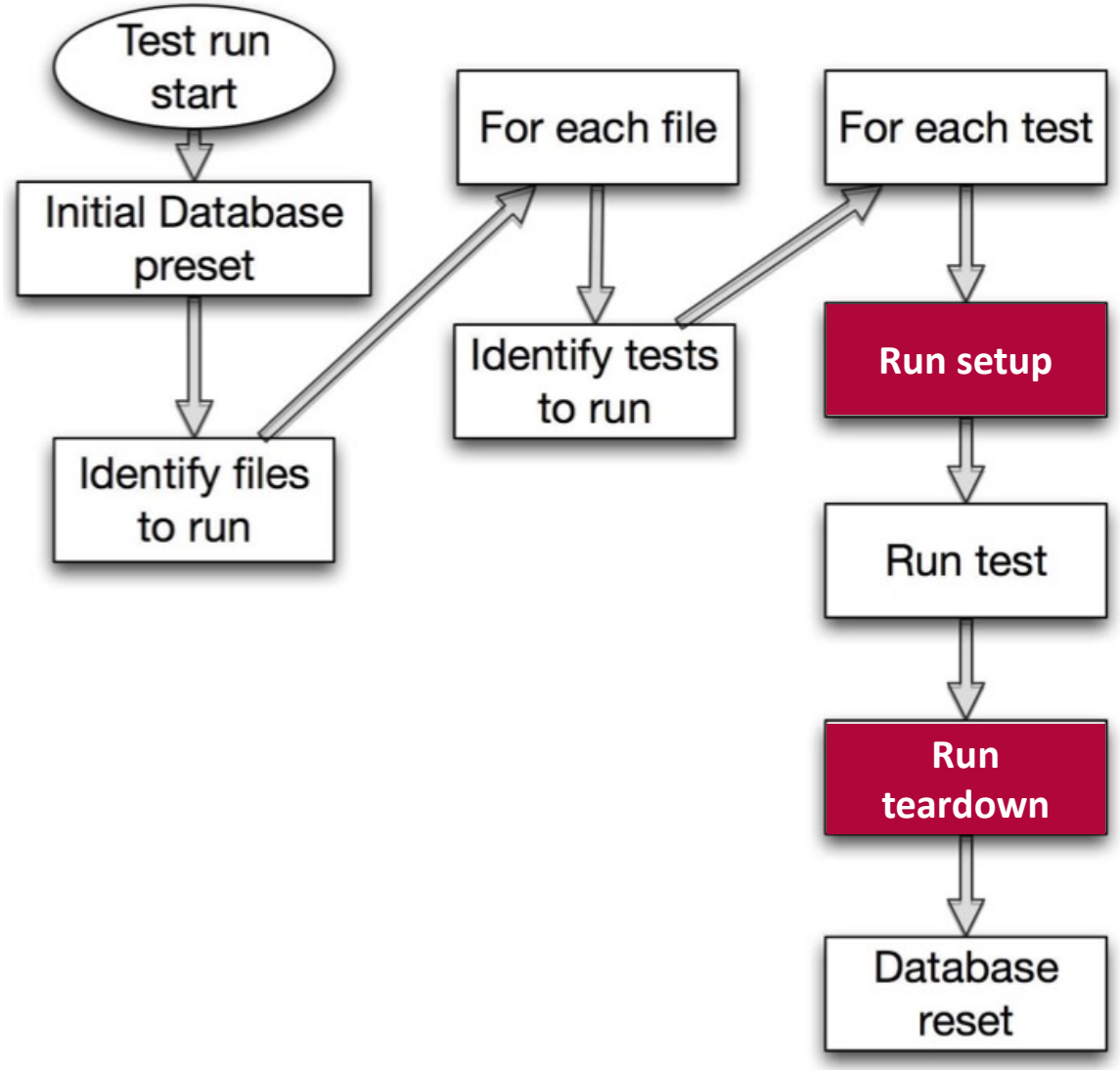
Setup and Teardown: RSpec

As a developer using RSpec
I want to execute code before and after test blocks
So that I can control the environment in which tests are run

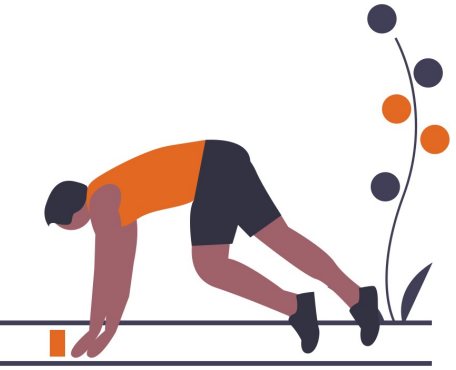
```
before(:each) # run before each test block
before(:all) # run one time only, before all of the examples in a group

after(:each) # run after each test block
after(:all) # run one time only, after all of the examples in a group
```

Typical Test Run



What do you expect is regularly done in setup and teardown steps?



Rails Test Prescriptions. Noel Rappin. 2010. p. 37. http://zepho.com/rails/books/rails-test-prescriptions.pdf

Agenda

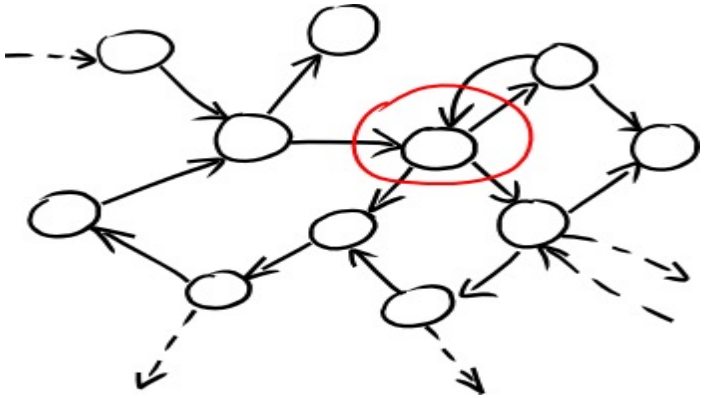
Advanced Concepts & Testing Tests

- Setup and Teardown
- **Test Data**
- Test Doubles

Isolation of Test Cases

Independent Tests

- Bug in a model should lead to **failures in tests related to this model only**
- Allow localization of bug

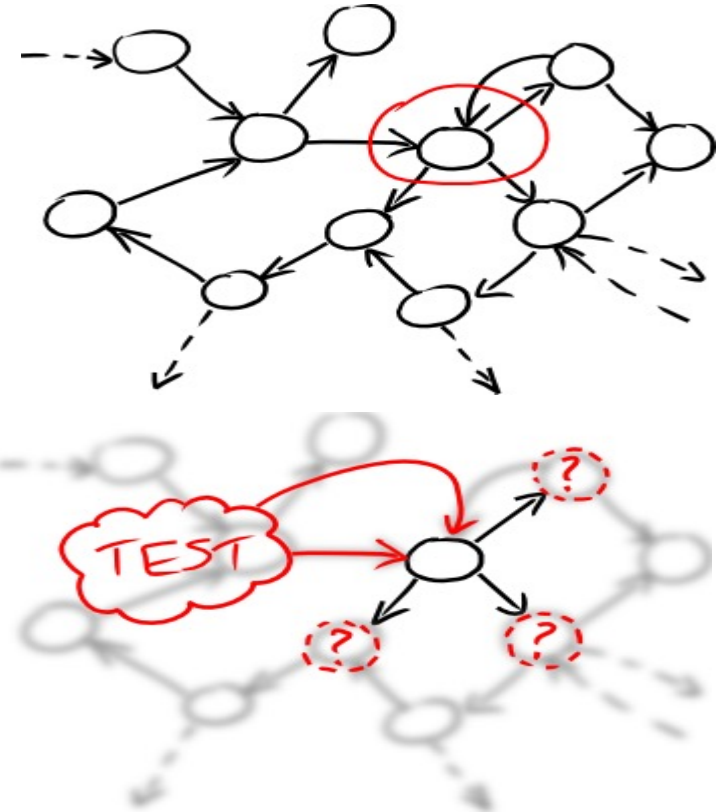


Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests

Isolation of Test Cases

Achieving Test Case Isolation

- Don’t write complex tests
- **Don’t share complex test data**
- Don’t use complex objects



Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests

Test Data Overview

Two main ways to provide data to test cases:

Fixtures

- Fixed state at the beginning of a test
- Assertions can be made against this state

Factories

- Blueprints for models
- Used to generate test data locally in the test

Fixture Overview

Fixtures for testing

- Fixed Sample data/objects
- Populate testing database with **predefined data** before test run
- Stored in database independent files (e.g. `test/fixtures/<name>.yml`)

```
# test/fixtures/users.yml
david: # Each fixture has a name
  name: David Heinemeier Hansson
  birthday: 1979-10-15
  profession: Systems development

steve:
  name: Steve Ross Kellock
  birthday: 1974-09-27
  profession: Front-end engineer
```

- http://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html
- http://guides.rubyonrails.org/testing.html

Drawbacks of Fixtures

Fixtures are global

- Only one set of data, every test has to deal with all test data

Fixtures are spread out

- Own directory
- One file per model -> data for one test is spread out over many files
- Tracing relationships is challenging

Fixtures are distant

- Fixture data is not immediately available in the test
- `expect(users(:ernie).age + users(:bert).age).to eq(20) #why 20?`

Fixtures are brittle

- Tests rely on fixture data, they break when data is changed
- Data requirements of tests may be incompatible

Test Data Factories

Test data should be

- **Local**: Defined as closely as possible to the test
- **Compact**: Easy and quick to specify; even for complex data sets
- **Robust**: Independent from other tests

One way to achieve these goals: **Data factories**

Defining Factories

```ruby
# This will guess the User class
FactoryBot.define do
  factory :user do
    first_name { "John" }
    last_name { "Doe" }
    admin false
  end

  # This will use the User class
  # (Admin would have been guessed)
  factory :admin, class: User do
    first_name { "Admin" }
    last_name { "User" }
    admin true
  end
end
```

Rails library: FactoryBot

- Rich set of features around
  - Creating objects
  - Connecting objects
- Rails automatically loads `spec/factories.rb` and `spec/factories/*.rb`

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Using Factories

- Different strategies: *build, create* (standard), *attributes_for*

```
# Returns a User instance that's _not_ saved
user = build(:user)

# Returns a _saved_ User instance
user = create(:user)

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

# Passing a block will yield the return object
create(:user) do |user|
  user.posts.create(attributes_for(:post))
end
```

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Factories: Attributes


```ruby
# Lazy attributes
factory :user do
  activation_code { User.generate_activation_code }
  date_of_birth { 21.years.ago }
end

# Dependent attributes
factory :user do
  first_name { "Joe" }
  email { "#{first_name}.#{last_name}@example.com".downcase }
end

# override the defined attributes by passing a hash/dict
create(:user, last_name: "Doe").email
# => "joe.doe@example.com"
```

The opposite of lazy is eager evaluation

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Factories: Associations


```
factory :post do
  # specify a different factory or override attributes
  association :author, factory: :user, last_name: ”Different“
End

# Builds and saves a User and a Post
post = create(:post)
post.new_record?              # => false
post.author.new_record?       # => false

# Builds and saves a User, and then builds but does not save a Post
post = build(:post)
post.new_record?              # => true
post.author.new_record?       # => false (why is this required?)
```

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Factories: Inheritance & Sequences


```ruby
# The title attribute is required
factory :post do
  Title { "A title" }
End

# Approved posts include an extra field
factory :approved_post, parent: :post do
  approved true
end
```

```ruby
# In-line sequence for a factory
factory :user do
  sequence(:email) {|n| “u#{n}@example.com"}
end
```

Custom code can be injected via „callbacks“

- http://www.rubydoc.info/gems/factory_bot/file/GETTING_STARTED.md

Agenda

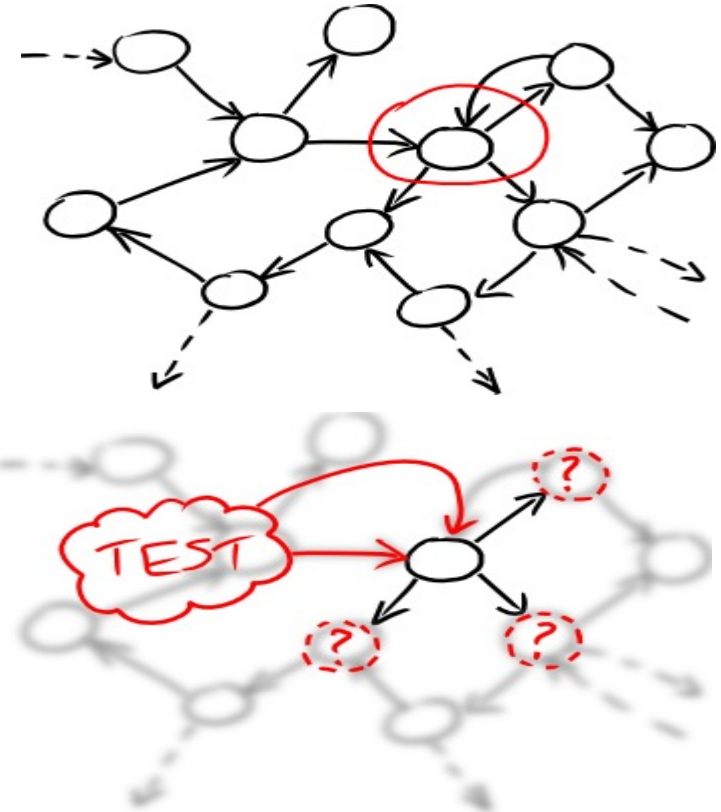
Advanced Concepts & Testing Tests

- Setup and Teardown
- Test Data
- **Test Doubles**

Isolation of Test Cases

Achieving Test Case Isolation

- Don't write complex tests
- Don’t share complex test data
- **Don’t use complex objects**



Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests

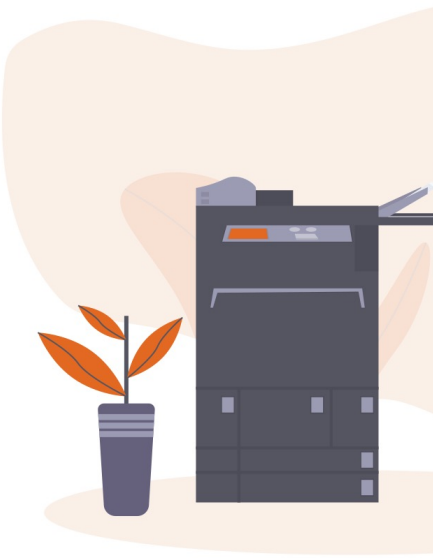
Test Doubles

Objects that stand in for the real thing in a test

- **Generic term** for range of testing techniques, think “stunt double”
- Purpose: simplify automated testing

Used when

- Real object is unavailable
- Real object is difficult to access or trigger
- Real object is **slow or expensive to run**
- An application state is required that is challenging to create



Ruby Test Double Frameworks

Many (Ruby) frameworks available:

- RSpec-mocks (http://github.com/rspec/rspec-mocks)
- Mocha, FlexMock

➡ We recommend **RSpec-Mocks.** Shares syntax with RSpec

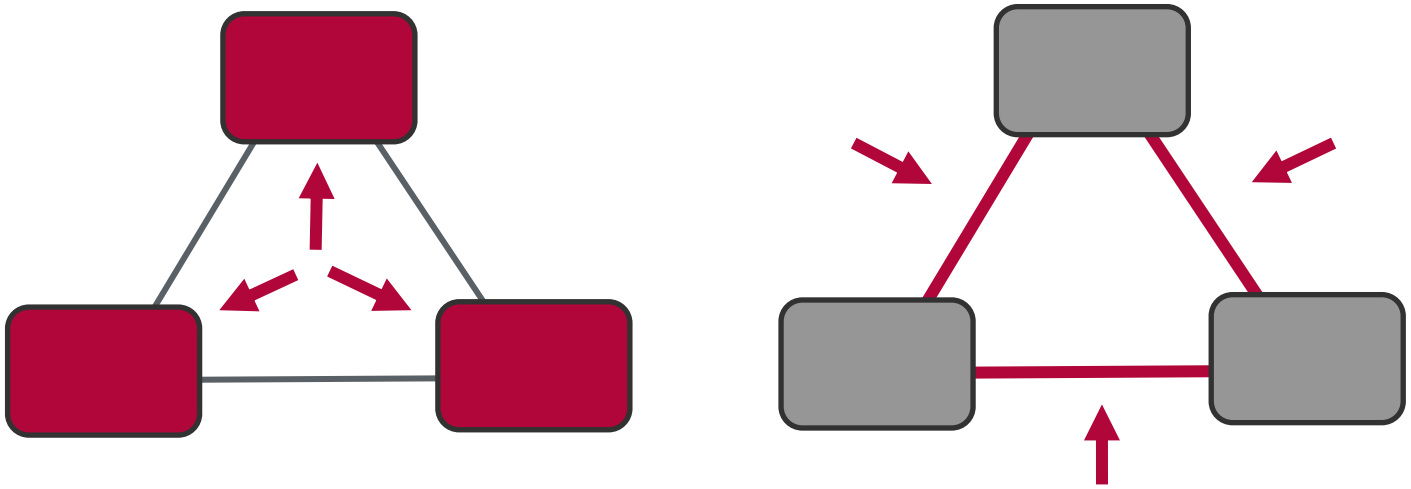
```
require("rspec/mocks/standalone")
```
imports the mock framework.
Useful for exploring in *rails console*.

Overview: https://www.ruby-toolbox.com/categories/mocking

Possibilities of Test Doubles

Verify behavior during a test

- Usually: **test system state after a test**
  - Only result of code are tested
  - Intermediate steps not considered
- **Test doubles**: Allow testing **detailed system behavior**
  - E.g. How often a method is called, in which order, with which parameters



Stub Testing Technique

Stubs: Provide canned responses to specified messages

- **Returns predefined value** if called
- No method call on the real object
- Strict by default (error when messages received that have not been allowed)

```
dbl = double(“user”)
allow(dbl).to receive_messages( name: “Fred”, age: 21 )
expect(dbl.name).to eq(“Fred”) #not really a good test :)
dbl.height #raises error (even if original object had property)
```

- Alternatively, if all method calls should succeed: **Null object double**

```
dbl = double(“user”).as_null_object
dbl.height.in_cm # this is ok! Returns itself (dbl)
```

- https://relishapp.com/rspec/rspec-mocks/v/3-10/docs/basics/null-object-doubles

Mock Testing Technique

Mocks: Define messages that must be received (or not received)

- Demands that mocked methods are called for test pass

```
book = double("book", title: "The RSpec Book")
expect(book).to receive(:open).once # 'once' is default
book.open # this works
book.open # this fails
```

- Or as often as desired

```
user = double("user")
expect(user).to receive(:email).exactly(3).times
expect(user).to receive(:level_up).at_least(4).times
expect(user).to receive(:notify).at_most(3).times
```

- If test ends with expected calls missing, it fails!

➡ **Mocks are stubs with attitude, mocks can fail tests**

- https://relishapp.com/rspec/rspec-mocks/v/3-10/docs/basics/expecting-messages

Spy Testing Technique

Spies: Record received messages, then assert they have been received

- Alternate way of using test doubles in *Given-When-Then* structure
- Allows **asserting that messages have been received at the end of test**

```
dbl = double(“user”).as_null_object # same as spy("user")
dbl.height
dbl.height
expect(dbl).to have_received(:height).at_least(2).times
```

This pattern for tests is also called **arrange-act-assert**

- Alternatively: spy on specific messages of real objects (partial doubles)

```
user = User.new
allow(user).to receive(:height)          # Given a user
user.measure_size                        # When I measure the size
expect(user).to have_received(:height) # Then height is called
```

- https://relishapp.com/rspec/rspec-mocks/v/3-10/docs/basics/spies
- https://thoughtbot.com/blog/a-closer-look-at-test-spies

Partial Test Doubles

Extension of real object instrumented with test-double behavior

- Mix real object and stubbed/mocked methods
- Only expensive methods might need replacing

```
s = "a user name" # s.length == 11
allow(s).to receive(:length).and_return(9001)
expect (s.length).to eq(9001) # the method was stubbed
s.capitalize! # this still works, only length was stubbed
```

- https://relishapp.com/rspec/rspec-mocks/v/3-10/docs/basics/partial-test-doubles

Expecting and Raising Errors

Testing exception handling

- A test double can raise an error when it receives a message
- Real error can be hard to provoke
- Test various types of exceptions: `and_raise(ExceptionClass)`

```
dbl = double()
allow(dbl).to receive(:foo).and_raise("boom")
dbl.foo # This produces:

#     Failure/Error: dbl.foo
#     RuntimeError:
#       boom
```

- https://relishapp.com/rspec/rspec-mocks/v/3-10/docs/configuring-responses/raising-an-error

Verifying Doubles

Check that methods being stubbed are present on underlying object

- Stricter alternative to normal doubles
- Confidence that doubles are not a complete fiction
- Verify that provided arguments are supported by method signature

```
class Post
  attr_accessor :title, :author, :body
end

post = instance_double("Post") # reference to the class Post
allow(post).to receive(:title)
allow(post).to receive(:message).with (‘a msg’) # this fails (not defined)
```

- https://relishapp.com/rspec/rspec-mocks/v/3-10/docs/verifying-doubles/using-an-instance-double

Test Doubles Pro and Contra

Disadvantages

- Test doubles must **accurately model real object behavior**
- Risk testing values set by test doubles
- Run out of sync with real implementation, brittle while refactoring

Advantages

- Allow tests focused on behavior
- Speed (e.g. not having to use an expensive database query)
- Isolation of tests

Best practice: try to minimize the amount of test doubles

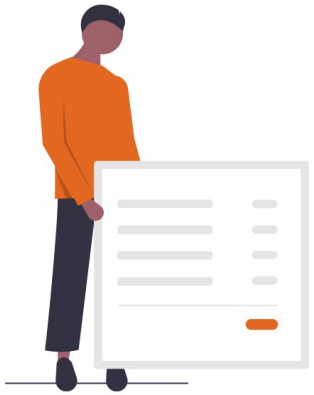
Summary

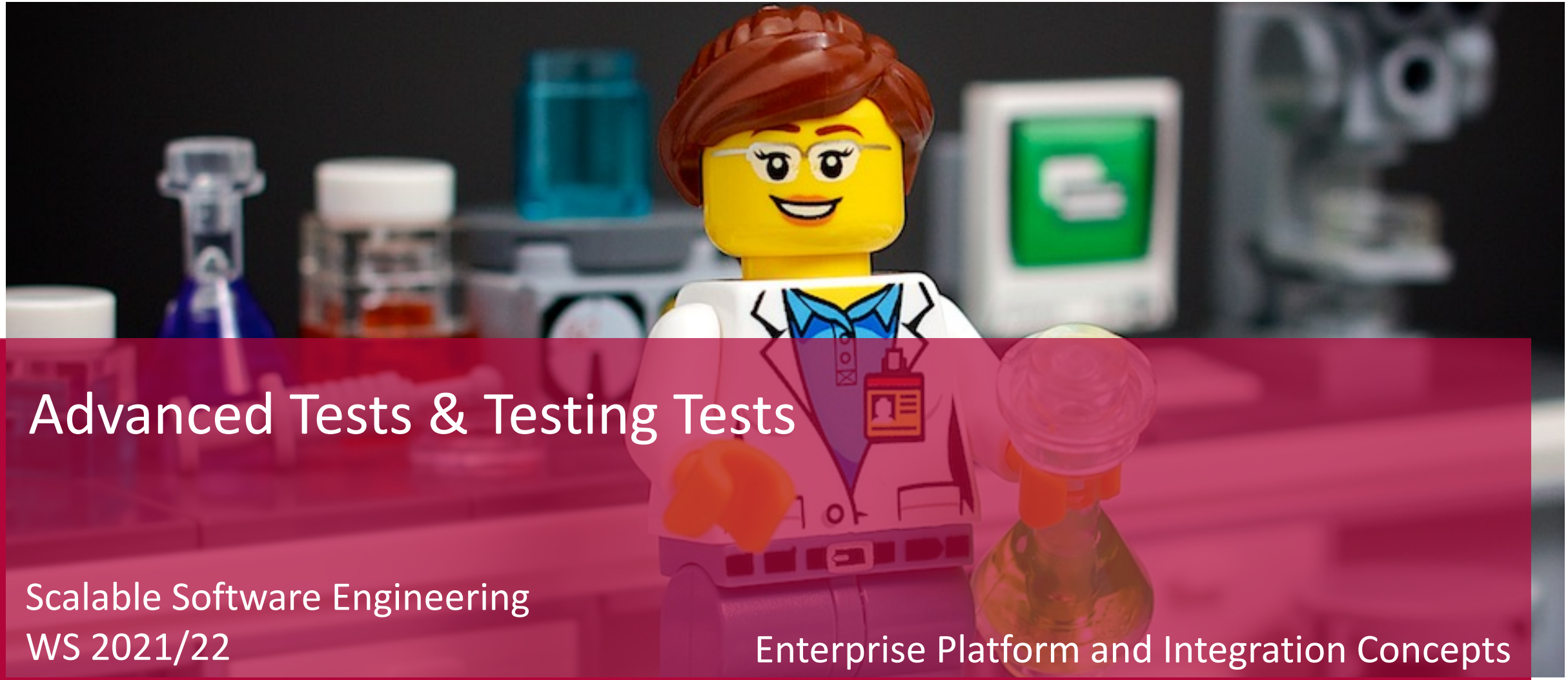
Test run steps

- Setup & teardown
- Test run process
- Test Data
  - Guiding principles
  - Fixtures vs factories

Test doubles

- Use cases & goals
- Mocks
- Stubs
- Spy
- Pros & Cons





Advanced Tests & Testing Tests

Scalable Software Engineering
WS 2021/22

Enterprise Platform and Integration Concepts

Code Coverage

Most commonly used metric for evaluating test suite quality

- Test coverage = executed code during test suite run ÷ all code * 100
  - e.g. 85 loc / 100 loc = 85% test coverage

Line coverage

- Absence of line coverage indicates potential problems
- **(High) line coverage means very little**
- In combination with good testing practices, coverage might say something about test suite reach
- Circa 100% test coverage is a by-product of BDD

Measuring Code Coverage

Different approaches to measure code coverage

- Line coverage
- Branch coverage

Tools

- SimpleCov: coverage tool for Ruby
- Uses line coverage by default

```
if (i > 0); i += 1 else i -= 1 end
```

➡ 100% line coverage even if one branch is not executed

Test Tips

Independence

- Of external test data
- Of other tests (and test order)

Repeatability

- Same results each test run
- Potential Problems
  - Dates, e.g. Timecop (https://github.com/travisjeffery/timecop)
  - Random numbers
  - Type and state of test database
  - Type of employed library depending on system architecture

Test Tips

Clarity

- Test **purpose should be immediately clear**
- Tests should be small, simple, readable
- Make it clear how the test fits into the larger test suite

Worst case:

```
it "sums to 37" do
  expect(37).to eq(User.all_total_points)
end
```

Better:

```
it "rounds total points to nearest integer" do
  User.add_points(32.1)
  User.add_points(5.3)
  expect(User.all_total_points).to eq(37)
end
```

Test Tips

Conciseness

- Use the minimum amount of code and objects
- But: Clear beats short
- Writing the minimum required amount of tests for a feature

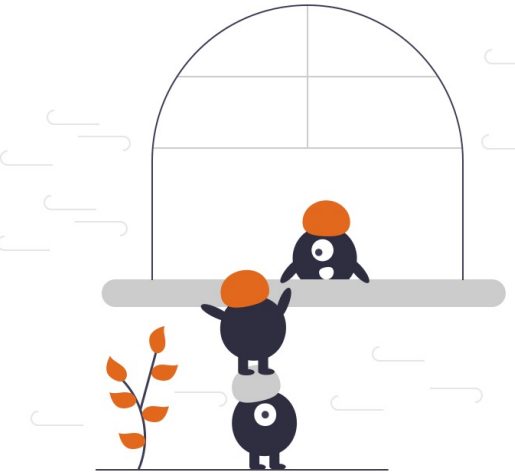
-> Test suite will be faster

```
def assert_user_level(points, level)
  user = User.create(points: points)
  expect(level).to eq(user.level)
end

it test_user_point_level
  assert_user_level(   0, "novice")
  assert_user_level(   1, "novice")
  assert_user_level( 500, "novice")
  assert_user_level( 501, "apprentice")
  assert_user_level(1001, "journeyman" )
  assert_user_level(2001, "guru")
  assert_user_level( nil, "novice")
end
```


Rails Test Prescriptions. Noel Rappin. 2010. p. 277. http://zepho.com/rails/books/rails-test-prescriptions.pdf

Conciseness: #Assertions per Test



If a single model method call results in many model changes:

- High number of assertions **->** High clarity and cohesion
- High number of assertions **->** Low test independence

- **Use context & describe and have few assertion per test**

Test Tips

Robustness

- Underlying code is correct -> test passes
- Underlying code is wrong -> test fails
- *Example:* view testing

```
describe "the signin process", type: :feature do
  it "signs me in (text version)" do
    visit '/dashboard'
    expect(page).to have_content “My Projects”
  end
  # version below is more robust against text changes
  it "signs me in (css selector version)" do
    visit '/dashboard'
    expect(page).to have_css "h2#projects"
  end
end
```

Test Tips

Robustness

- Reusable code increases robustness
- E.g. constants instead of magic numbers

```
def assert_user_level(points, level)
  user = User.build(points: points)
  expect(user.level).to eq(level)
end

def test_user_point_level
  assert_user_level(User::NOVICE_THRESHOLD + 1, "novice")
  assert_user_level(User::APPRENTICE_THRESHOLD + 1, "apprentice")
  # ...
end
```

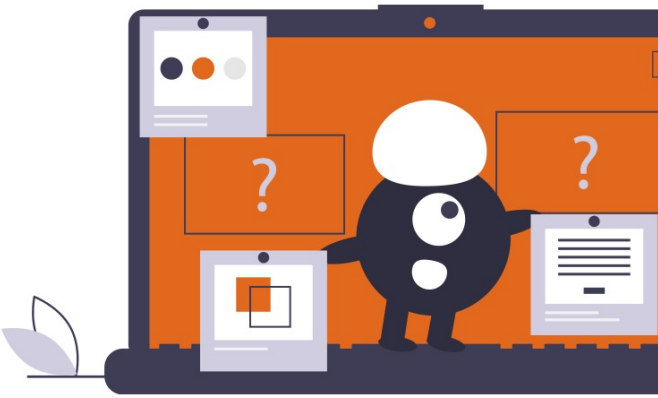
- Be aware of tests that always pass regardless of underlying logic

- Rails Test Prescriptions. Noel Rappin. 2010. p. 278. http://zepho.com/rails/books/rails-test-prescriptions.pdf

Manual Fault Seeding

Conscious introduction of faults into the program

- Run tests
- Minimum 1 test should fail



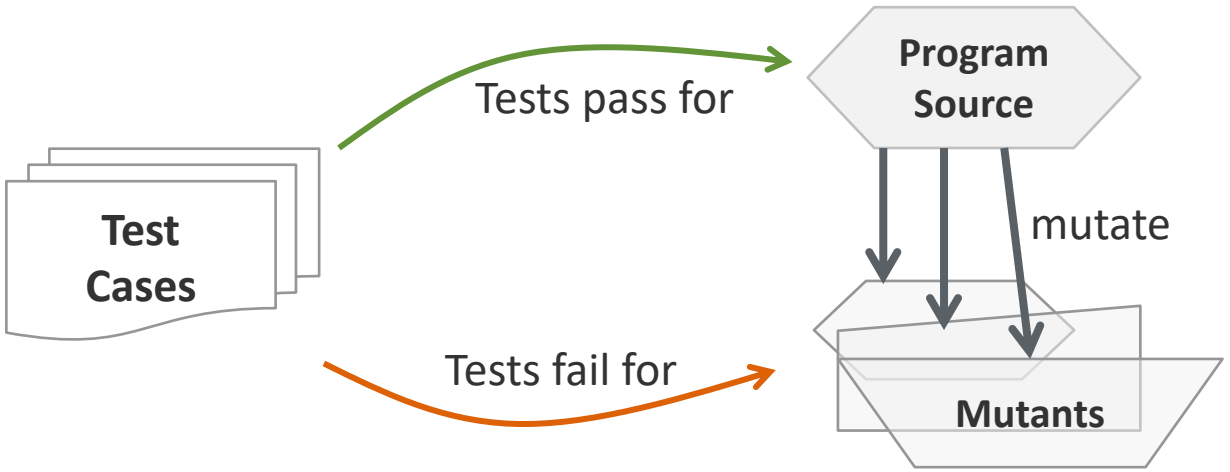
If no test fails, then a test is missing

- Possible even with 100% line coverage
- Asserts functionality coverage

Mutation Testing

Mutant: Modified version of the program with small change

- Tests correctly cover code -> Test should notice change and fail



next_month:

```
if month > 12 then
  year += month / 12
  month = month % 12
end
```

```
if not month > 13 then
  year -= month / 12
  month = month % 12
end
```

- **Mutation Coverage**: How many mutants did not cause a test to fail? Asserts functionality & behavior coverage
  - For Ruby: *Mutant* (https://github.com/mbj/mutant)

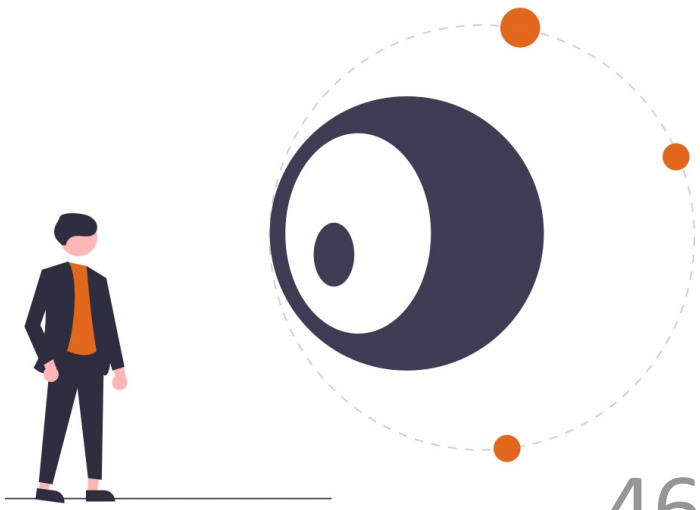
Metamorphic Testing

When testing, often hard to find test oracle

- Establish whether a test has passed or failed
- Require understanding of exact input-output-relation
- May be more convenient to **reason about relations between outputs**

Compare outputs of system-under-test

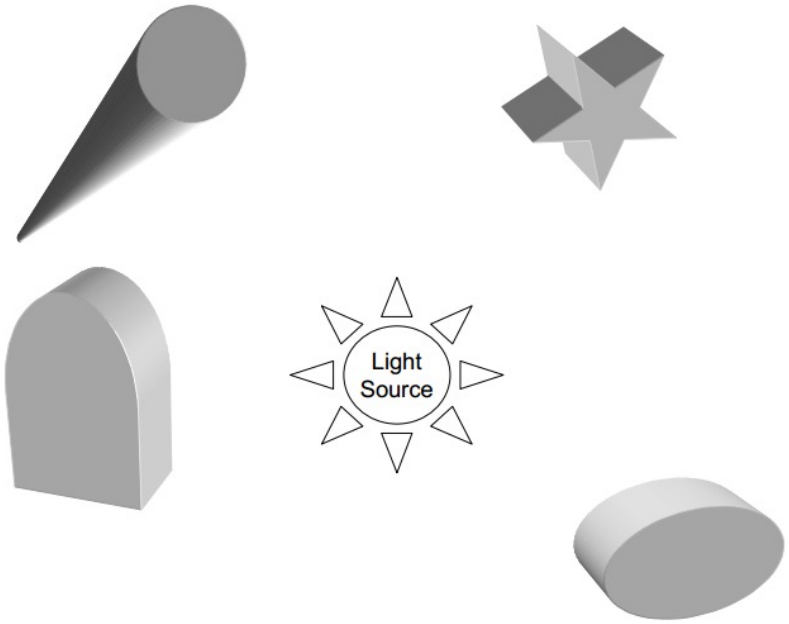
- Describe inherent behavior of the program
- No need to know exact outputs (in advance)
- Test the invariants: f(x) < f(x+1)



Metamorphic Testing Example

Scenario: Rendering lighting in a digital scene

- Hard to verify all pixels have correct color
- **Use relations of outputs for test cases**



Test: Position of light source changes

- Points closer to light source will be brighter
  - Exception: White pixels
- Points further away from light source will be darker
  - Exception: Black pixels
- Points hidden behind other objects don't change brightness

Fuzzing / Fuzz Testing

Automated software testing technique

- Provide randomized or invalid inputs to a program
- Capture exceptions, e.g. crashes, failing assertions, or memory leaks
- Expose corner cases that are unhandled

Program inputs

- Input that passes the input parser, but is strange enough for unusual behavior
- Input that crosses a system boundary, e.g. user input or network packets

Further Reading

- http://betterspecs.org – Collaborative RSpec best practices documentation effort
- *Everyday Rails Testing with RSpec* by Aaron Sumner, leanpub
- *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends* by David Chelimsky et al.

Summary

Advanced Tests & Testing Tests

- Test Coverage
- Test Tips
- Fault Seeding
- Mutation Testing
- Metamorphic Testing