Mocha

Ein Ruby Mocking und Stubbing Framework

von Thilo Utke

Werd ich drüber reden
Erst was zu mit.

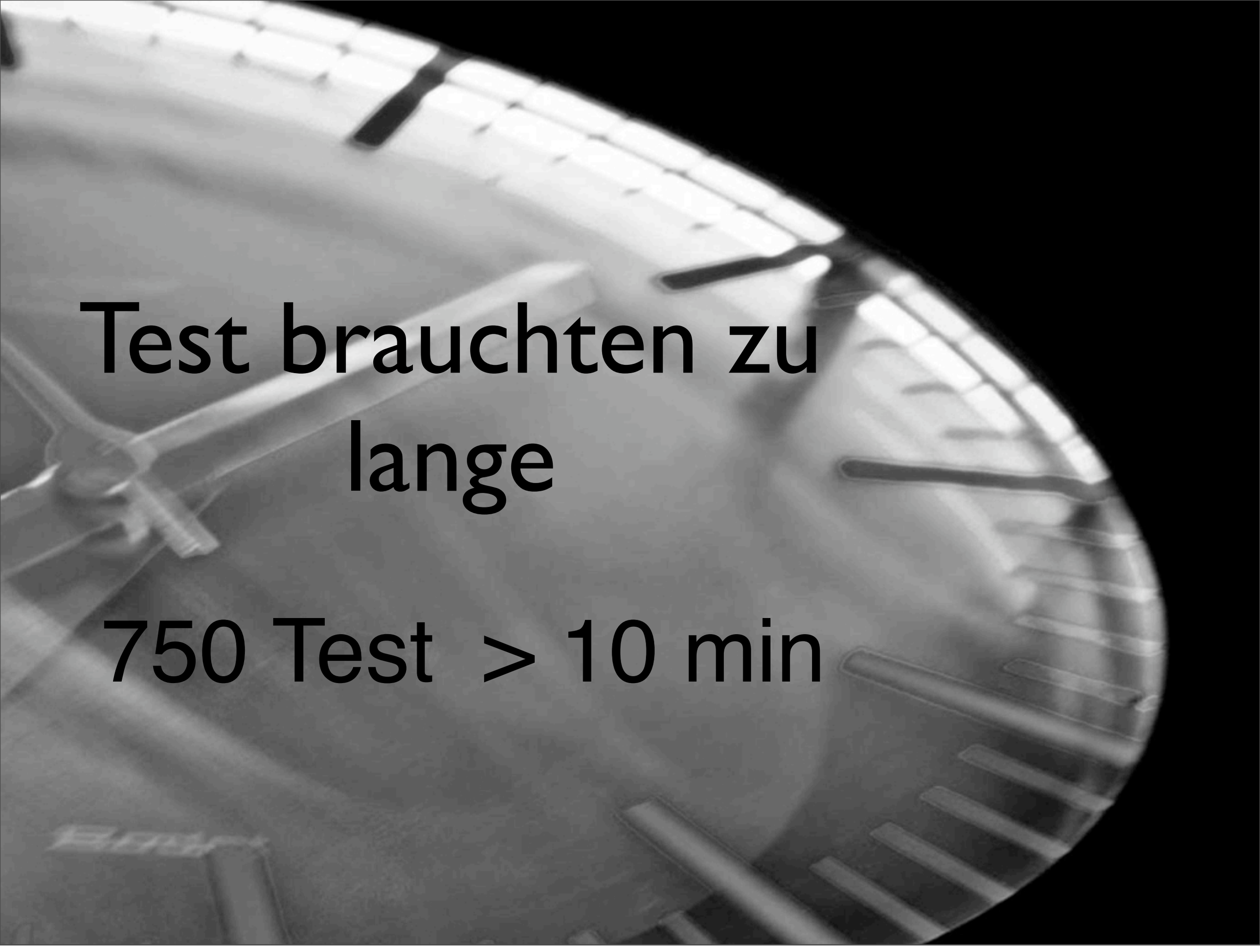
- 4 Jahre Software Entwicklung
- 3 Jahre .Net
- 1 Jahr Ruby/Rails
- 1 Jahr Selbständig
- <1 Jahr Gründer

Zusammen mit Alex.

Testgetriebe Entwicklung

Autoki: Tests > 1200

Testgetrieben: Tests spezifizieren Funktion


Test brauchten zu
lange
750 Test > 10 min

Unteranderem Deshalb:

5 DB Schreibzugriffe
5 Objekterzeugungen
5 Validierungen

```
def test_index_shows_logg

  5.times { create_user :game_points => 10 }
  u = create_logged_in_user :game_points => 3
  get :index
  assert_select '#hall_of_fame td', u.name
end
```

Lösungen

- Fixtures
- Inmemory DB
- Stubs/Mocks
- ~~Tests Reduzieren~~

Fixtures - Sind für alle Tests gleich, total instabiel.
InMemory DB - Kann mysql nicht, extra Setupaufwand
Rails bringt Mocks mit - Extra Programmieraufwand.
Test reduzieren - Kommt nicht in Frage!

Vorteile Stubs/Mocks

- Flexiebel
- Sanfter Übergang

Dummy Objekte

```
nil
Objekt.new
```

Stub

- Nur Methoden für Test
- Feste Rückgabewerte

```
class Cal
   def today
      Date.now
   end
end

class CalStub
   def today
      Date.parse ’20.12.1995’
   end
end
```

Test Spy

- Zeichnet Methodenaufrufe auf
- Prüft Ergebnisse am TestSpy nach dem Aufruf

```ruby
class Horn

  def honk
    # hup, hup!
  end

end

class Auto

  def initialize(horn = Horn.new)
    @horn = horn
  end

  def honk
    @horn.honk
  end

end
```

```
class HornSpy

  attr_reader :number_of_calls

  def initialize
    @number_of_calls = 0
  end

  def honk
    @number_of_calls += 1
  end

end

auto = HornSpy.new
auto = Auto.new(horn)
auto.honk

assert_equal 1, horn.number_of_calls
```

Mock

- Spezifiziert erwartete Methodenaufrufe
- Prüft ob Aufruf mit Spezifikation übereinstimmt

```ruby
class Horn

  def honk(times)
    # ring, ring
  end

end

class Auto

  def initialize(horn1 = Horn.new, horn2 = Horn.new)
    @horn1 = horn1 ;@horn2 = horn2
  end

  def fanfare
    @horn1.honk(3); @horn2.honk(1)
  end

end
```

```ruby
class MockHorn
  include Test::Unit::Assertions

  def initialize(expected_calls)
    @expected_calls = expected_calls
    @actual_calls = 0
  end

  def honk(time)
    @actual_calls += times
  end

  def verify
    assert_equal @expected_calls, @actual_calls
  end

end
```

```
horn1 = MockHorn.new(3)
horn2 = MockHorn.new(1)
auto = Auto.new(horn1, horn2)

auto.fanfare

horn1.verify
horn2.verify
```

Ist ne Menge Schreibarbeit das alles selber zu mocken/stubben.

Framework: Mocha

- Leichte Erstellung Mocks/Stubs
- Kann mit echten Objekten arbeiten
- Leichte Syntax

Leichte Syntax - ist gleich für Mocken/Stuben bei Mocks und auf Objekten

Vorher

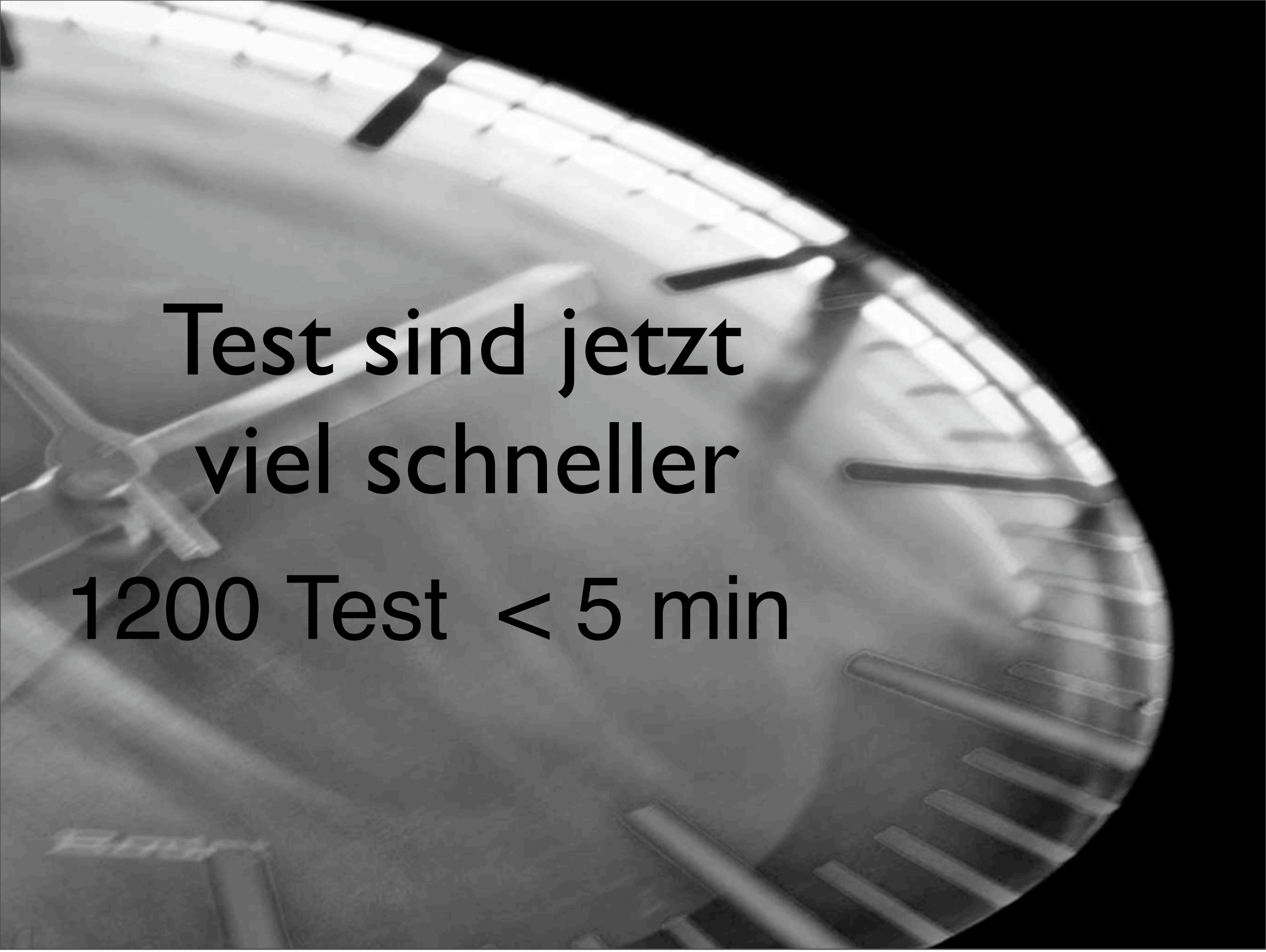
5 DB Schreibzugriffe
5 Objekterzeugungen
5 Validierungen

```
def test_index_shows_log      _in_user
  5.times { create_user :game_points => 10 }
  u = create_logged_in_user :game_points => 3
  get :index
  assert_select '#hall_of_fame td', u.name
end
```

Nachher

0 DB Schreibzugriffe
1 Objekterzeugungen
0 Validierungen

```
def test_index_shows_logged
  user = new_logged_in_us      game_pe
  Game.expects(:all_time_rank_of_user).with(user).returns(1)
  get :index
  assert_select '#hall_of_fame td', user.name
end
```


Test sind jetzt
viel schneller
1200 Test < 5 min

Offensichtliche Vorteile

- Geringere Kopplung
- Schneller
- Weniger Datenbankzugriffe
- Innere Zustände müssen nicht zum Testen freigelegt werden.

Offensichtliche Nachteile

- Fehlende Funktionen werden gestubbed/ gemocked und vergessen zu implementieren.
- Aufwand fürs Testschrieben ist grösser

Functional Tests, damit eben keine Schnittstellenaufrufe vergessen werden

Und wem das jetzt zu wenig Code war...

Objekte stubben

```
test_auto_name_includes_model
    model = stub('model', :name => 'a4')
    auto = Auto.new(model)
    assert auto.name.include?(model.name)
end
```

Instanzmethoden stubben

```
test_auto_save_increases_version
    auto = Auto.new(:version = 1)
    auto.stubs(valid?).returns(true)
    auto.save
    assert_equal 2, auto.version
end
```

Klassenmethoden stubben

```
test_show_shows_auto
    auto = stub(:name => "Test Auto")
    Auto.stubs(:find).returns(auto)
    get :show, :id => 1
    assert_response :success
end
```

Hier mal ein Rails Controller Test

Insanzvariablen stubben

```
test_save_fails_if_lockversion_is_wrong
    auto = Auto.new
    auto.instance_variable_set(:@lock_version, 1)
    assert !auto.save
end
```

Noch nie benötigt.

Instanzmethoden für alle Instanzen stubben

```
test_auto_save_increases_version
   Auto.any_instance.stubs(valid?).returns(true)
   auto = Auto.new(:version = 1)
   auto2 = Auto.new(:version = 2)
   auto.save
   auto2.save
   assert_equal 2, auto.version
   assert_equal 3, auto2.version
end
```

Für jedes neu erzeugte Objekt ist die Methode gestubbed.

Objekte mocken

```
test_auto_name_uses_model_name
    model = mock('model', :name => 'a4')
    auto.new(model)
    auto.name
end
```

Mockinjection via Constructor.

Methoden mocken

```
test_save_does_noting_if_not_own_auto
    auto = stub()
    Auto.stubs(:find).returns(:auto)
    auto.expect(:save).never
    get :save, :id => 1
  end
```

Für Klassen lässt sich auch expects so aufrufen.
.never/.times(2)/at_leatst... gibt an wie oft die Schnittstelle aufgerufen werden darf/muss/kann.

Parametermatcher

```
test_play_sets_game_state_to_next_turn
    game = stub(:state => 'open')
    game.expect(:state=).with('next_turn')
    game.play
end
```

Um spezifische Parameter beim Aufrufen zu testen.
Es gibt noch zahlreiche Varianten mehr....

Mocha Fallstricke

- Wenn Mocks unerwartete Aufrufe erhalten kann man ganz schön suchen (z.B. bei find)
- any_instance ist nicht richtig isoliert, wirkt sich auf Tests mit echten Objekten aus.

any_instance kann durch stubben von .new umgangen werden.

Wann mocken?
Wann stubben?

Eher weniger Mocken. Ojekteveränderungen mocken. Queries stubben.
Oder:
Alles was direkt mit dem Tesfall zusammenhängt mocken, externe Abhängigkeiten stubben.

Mocha

http://mocha.rubyforge.org/

Folien und weitere Links auf:
upstream-berlin.com