

Behavior-Driven Development with RSpec

Hosted by Pat Maddox
Professional Ruby Conference 2008

Before we begin...

- `sudo gem install rspec-rails`
- `sudo gem install cucumber`
- `git clone`
`git://github.com/pat-maddox/blackjack.git`

Why are you here?

- You need another *DD in your life (BDD, in our case)
- You're interested in testing
- You want to know how RSpec will make your life better

Why am I here?

- I've been using RSpec daily for over 2 years
- Core team member, and author of a way-late book from AW
- I love talking about this stuff!!!!

What will you learn?

- RSpec's major components
- BDD rhythm
- Acceptance testing with Cucumber
- “Outside-in”

Two Underlying Goals

- Clean code that works
- Clean software that works

PROFIT

Lots of Kinds of Tests

- We care about
 - Developer Tests
 - Customer Tests

Developer Tests

- We own them
- Verify that our code works right
- Design tool
- Unit Tests
- Integration Tests

Customer Tests

- Customer owns them
 - with our collaboration
- Verify that our code does the right work
- Acceptance Tests

RSpec: 4 frameworks in 1

- Expectations framework
- Example framework
- Mock objects framework
- Acceptance testing (provided by Cucumber)

Expectations Framework

- Expectations: verification
- Matchers: what to verify
- Natural language-ish
- Focus on behavior

```
>> require 'spec'
=> true
>> include Spec::Expectations
=> Object
>> 1.should == 1
=> true
>> 1.should_not == 2
=> true
>> 1.should == 2
Spec::Expectations::ExpectationNotMetError: expected: 2,
  got: 1 (using ==)
    from /Library/Ruby/Gems/1.8/gems/rspec-1.1.11/lib/spec/expectations.rb:5
2:in `fail_with'
    from /Library/Ruby/Gems/1.8/gems/rspec-1.1.11/lib/spec/matchers/operator
_matcher.rb:46:in `fail_with_message'
    from /Library/Ruby/Gems/1.8/gems/rspec-1.1.11/lib/spec/matchers/operator
_matcher.rb:61:in `__delegate_method_missing_to_given'
    from /Library/Ruby/Gems/1.8/gems/rspec-1.1.11/lib/spec/matchers/operator
_matcher.rb:12:in `=='
    from (irb):5
>> █
```

Expectations & Matchers

- `l.should_not == 2`
- `team.should have(1 1).players`
- `lambda { 100/0 }.should raise_error`

```
>> require 'spec'
=> true
>> include Spec::Expectations
=> Object
>> include Spec::Matchers
=> Object
>> o = Object.new; def o.example?; true; end
=> nil
>> o.should be_example
=> true
>> [1,2,3,4,5].should have(5).items
=> true
>> lambda { raise "oh noez!" }.should raise_error(/noez/)
=> true
>> a = [:foo, :bar, :baz]
=> [:foo, :bar, :baz]
>> lambda { a.delete(:bar) }.should change(a, :size).by(-2)
Spec::Expectations::ExpectationNotMetError: size should have been changed by -2,
  but was changed by -1
    from /Library/Ruby/Gems/1.8/gems/rspec-1.1.11/lib/spec/expectations.rb:5
2:in `fail_with'
    from /Library/Ruby/Gems/1.8/gems/rspec-1.1.11/lib/spec/expectations/hand
ler.rb:17:in `handle_matcher'
    from /Library/Ruby/Gems/1.8/gems/rspec-1.1.11/lib/spec/expectations/exte
nsions/object.rb:31:in `should'
    from (irb):9
    from :0
>>
```

Custom Matchers

- You can define custom matchers if none of the built in ones suit you
- Standard custom matchers
- Simple matcher

```

module Spec
  module Matchers

    class BeClose #:nodoc:
      def initialize(expected, delta)
        @expected = expected
        @delta = delta
      end

      def matches?(given)
        @given = given
        (@given - @expected).abs < @delta
      end

      def failure_message
        "expected #{@expected} +/- (< #{@delta}), got #{@given}"
      end

      def description
        "be close to #{@expected} (within +/- #{@delta})"
      end
    end

    # :call-seq:
    #   should be_close(expected, delta)
    #   should_not be_close(expected, delta)
    #
    # Passes if given == expected +/- delta
    #
    # == Example
    #
    #   result.should be_close(3.0, 0.5)
    def be_close(expected, delta)
      Matchers::BeClose.new(expected, delta)
    end
  end
end
end

```

```

module Spec
  module Matchers

    # :call-seq:
    #   should be_close(expected, delta)
    #   should_not be_close(expected, delta)
    #
    # Passes if actual == expected +/- delta
    #
    # == Example
    #
    #   result.should be_close(3.0, 0.5)
    def be_close(expected, delta)
      simple_matcher do |actual, matcher|
        matcher.failure_message = "expected #{expected} +/- (< #{delta}), got #{actual}"
        matcher.description = "be close to #{expected} (within +/- #{delta})"
        (actual - expected).abs < delta
      end
    end
  end
end
end

```

Example Framework

- Specification, not verification
- Specification by example
- Developer testing / unit testing

```
it "a hotel should not be full when booked less than capacity" do
  99.times { @hotel.book_room }
  @hotel.should_not be_full
end

it "an account should have a lower balance when money is withdrawn" do
  a = Account.new(100)
  a.withdraw(25)
  a.balance.should == 75
end

it "should be a perfect game when 12 strikes are thrown" do
  12.times { game.roll(10) }
  game.should be_perfect
end
```

“Everyone knows that specific events have a vividness which imprints them so strongly on the memory that they can later be used as models for other events which are like them in some way. Thus in each specific event, there is the germ of a whole class of similar events.”

- Douglas Hofstadter, *GED*

Your brain understands examples

- `Math.sqrt(9).should == 3`
- `Math.sqrt(16).should == 4`
- `Math.sqrt(25).should == ??`

Example Groups

- Used for organizing examples
- Encourage you to keep code modular
- Final piece of specifying behavior - these are the full specification

```
describe Hotel do
  before(:each) do
    @hotel = Hotel.new(100)
  end

  it "should not be full when booked less than capacity" do
    99.times { @hotel.book_room }
    @hotel.should_not be_full
  end

  it "should be full when booked to capacity" do
    100.times { @hotel.book_room }
    @hotel.should be_full
  end

  it "should not allow booking when full" do
    lambda {
      @hotel.book_room
    }.should raise_error(OverCapacityError)
  end
end
```

- Provide `before()` callback for setting up context
- Provide `after()` callback for cleanup
- Can be nested to share context

```
describe Team, "offering a contract to a Player" do
  before(:each) do
    @team = Team.new
    @player = Player.new(:skill => 98)
  end

  describe "player is friendly" do
    before(:each) do
      @player.disposition = :friendly
    end

    it "should offer a contract worth $5 million salary" do
      @team.offer(@player).salary.should == 5_000_000
    end

    it "should offer a signing bonus of $1 million" do
      @team.offer(@player).bonus.should == 1_000_000
    end
  end

  describe "player is arrogant" do
    before(:each) do
      @player.disposition = :arrogant
    end

    it "should offer a contract worth $3 million salary" do
      @team.offer(@player).salary.should == 3_000_000
    end

    it "should offer no signing bonus" do
      @team.offer(@player).bonus.should == 0
    end
  end
end
```

Shared Example Groups

- Refactor your specs!
- Great for extracting commonalities in specs

```
describe "requires admin access", :shared => true do
  before(:each) do
    login_as joe_user
  end

  it "should redirect to the login page" do
    do_request
    response.should redirect_to(login_url)
  end
end

describe AccountsController do
  describe "PUT /accounts/1" do
    it_should_behave_like "requires admin access"
    def do_request; put :update, :id => 1 end
  end

  describe "DELETE /accounts/1" do
    it_should_behave_like "requires admin access"
    def do_request; delete :destroy, :id => 1 end
  end
end
```

Good examples are

- Clear
- Concise
- Isolated
- Fast

Simple Code

- Passes all the tests
- Reveals all the intention
- No duplication
- Fewest number of classes or methods

- Kent Beck

The BDD Cycle

red - green - refactor

Encapsulated Thinking

- Encapsulation - separating “what to do” from the details of “how to do it”
- BDD cycle
 - Red - what to do
 - Green - how to do it
 - Refactor - how to do it better

Red

```
class Dollar
  def initialize(value)
    @value = value
  end
end

describe Dollar do
  it "should print itself" do
    Dollar.new(525).to_s.should == '$5.00'
  end
end
```

```
~/code:$ ruby my_example_spec.rb
F

1)
'Dollar should print itself' FAILED
expected: "$5.00",
  got: "#<Dollar:0x507a30>" (using ==)
my_example_spec.rb:12:
my_example_spec.rb:10:

Finished in 0.006857 seconds

1 example, 1 failure
~/code:$
```

Green

```
class Dollar
  def initialize(value)
    @value = value
  end

  def to_s
    '$5.00'
  end
end

describe Dollar do
  it "should print itself" do
    Dollar.new(525).to_s.should == '$5.00'
  end
end
```

```
Terminal — bash — 80x24
~/code:$ ruby my_example_spec.rb
.

Finished in 0.007368 seconds

1 example, 0 failures
~/code:$
```

Refactor

```
class Dollar
  def initialize(value)
    @value = value
  end

  def to_s
    sprintf('%.2f', @value / 100.0)
  end
end

describe Dollar do
  it "should print itself" do
    Dollar.new(520).to_s.should == '$5.20'
  end
end
```

Red

```
class Account
  attr_reader :balance

  def initialize(balance)
    @balance = balance
  end

  def withdraw(amount)
    @balance -= amount
  end

  def deposit(amount)
    @balance += amount
  end
end

describe Account do
  it "should know the number of transactions that it has made" do
    a = Account.new(100)
    a.withdraw 10
    a.deposit 50
    a.num_transactions.should == 2
  end
end
```

```
~/code:$ ruby account_spec.rb
```

```
F
```

```
1)
```

```
NoMethodError in 'Account should know the number of transactions that it has made'
```

```
undefined method `num_transactions' for #<Account:0x50643c @balance=140>
```

```
account_spec.rb:25:
```

```
account_spec.rb:20:
```

```
Finished in 0.008419 seconds
```

```
1 example, 1 failure
```

```
~/code:$
```

Green

```
class Account
  attr_reader :balance

  def initialize(balance)
    @balance = balance
    @transactions = 0
  end

  def withdraw(amount)
    @transactions += 1
    @balance -= amount
  end

  def deposit(amount)
    @transactions += 1
    @balance += amount
  end

  def num_transactions
    @transactions
  end
end

describe Account do
  it "should know the number of transactions that it has made" do
    a = Account.new(100)
    a.withdraw 10
    a.deposit 50
    a.num_transactions.should == 2
  end
end
```

```
Terminal — bash — 80x24
~/code:$ ruby account_spec.rb
.

Finished in 0.007685 seconds

1 example, 0 failures
~/code:$
```

Refactor

```
class Account
  def initialize(balance)
    @transactions = [balance]
  end

  def balance
    @transactions.sum
  end

  def withdraw(amount)
    @transactions << amount
  end

  def deposit(amount)
    @transactions << amount
  end

  def num_transactions
    @transactions.size - 1
  end
end

describe Account do
  it "should know the number of transactions that it has made" do
    a = Account.new(100)
    a.withdraw 10
    a.deposit 50
    a.num_transactions.should == 2
  end
end
```

```
Terminal — bash — 80x24
~/code:$ ruby account_spec.rb
.

Finished in 0.007306 seconds

1 example, 0 failures
~/code:$
```

**Don't forget to
refactor!!!!!!**

Exercise: Feeling the Flow

- `git clone`
`git://github.com/pat-maddox/blackjack.git`
- `git co -b just_specs origin/just_specs`
- `ruby spec/hand_spec.rb` # or `autospec`

Acceptance Tests

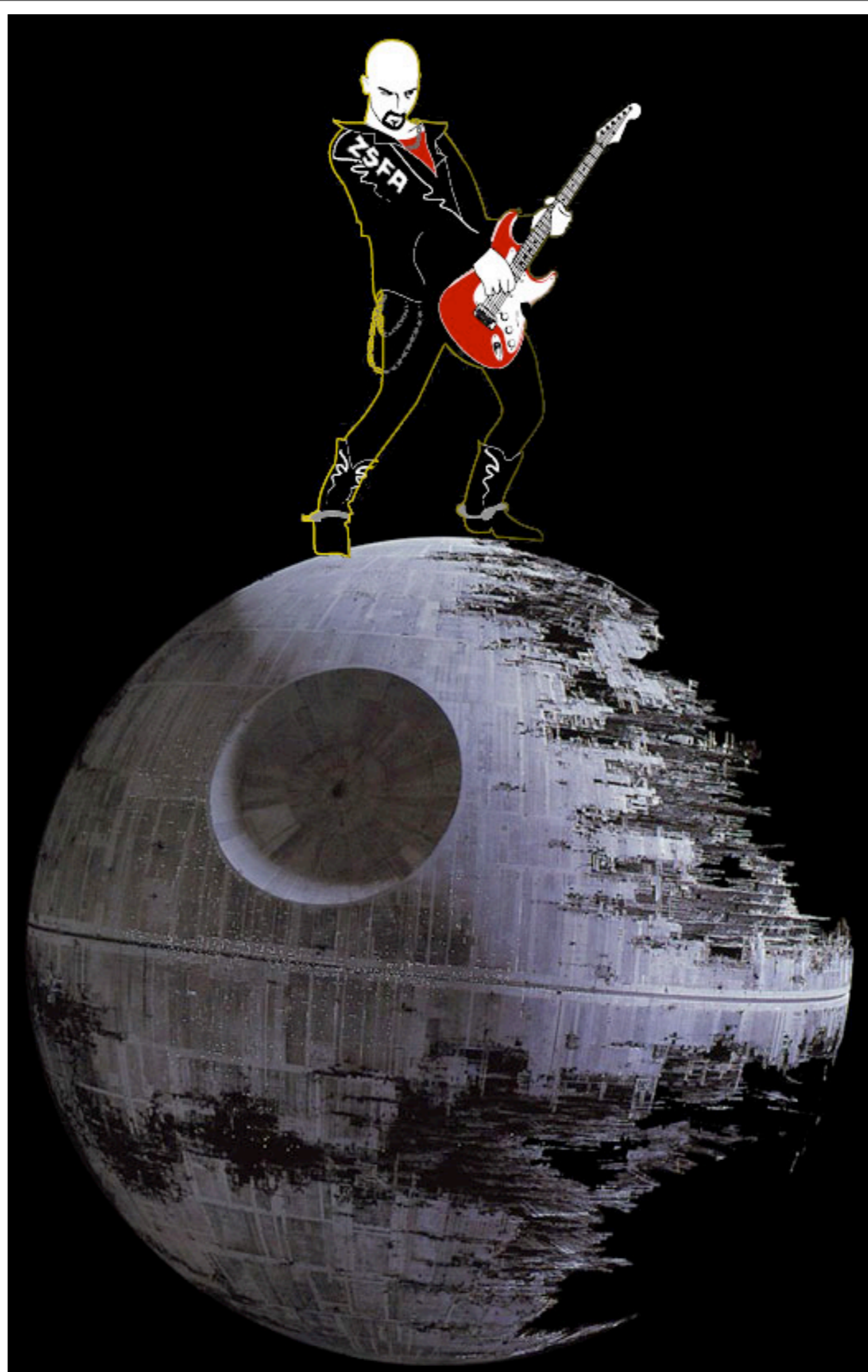
- Specifications of the system's behavior
- Written in domain terms
- Maintained by customer and developer

Clean Software

- Runs all the tests
- Focuses on the domain, hides infrastructure
- Every feature provides business value

Communication & Responsibility





System Specs

- Typically use the outer API of the system
 - execute binary for command line apps
 - web apps: in-browser, or HTTP requests
- Focus on inputs to the system, outputs and side-effects
 - account balance changes
 - email notification sent to admin

Implementation Levels

- Model
- Controller API
- Browser using Selenium/Watir

How to Write Good Stories

- Use domain language
 - Ubiquitous language - Evans, *DDD*
- No mention of infrastructure
- Imperative vs Declarative

Scenario: Customer chooses free shipping

Given a shopping cart total of \$100

When I check out

Then I should see an offer for free shipping

When I choose free shipping

And I submit the order

Then the order total should be \$100

Scenario: Customer chooses express shipping

Given a shopping cart total of \$100

When I check out

Then I should see an offer for free shipping

When I choose express shipping

And I submit the order

Then the order total should be \$125

Scenario: Customer checks out

Given a shopping cart total of \$100

When I press the submit button

Then the order should be in the database

BAD

Scenario: Customer checks out
Given a shopping cart total of \$100
When I press the submit button
Then the order should be in the database

Imperative vs Declarative

- Imperative
 - Fine-grained
 - Coupled to implementation
- Declarative
 - Coarse-grained
 - Less coupled

Imperative

```
Scenario: Customer registers for the first time
  Given I am at the new user signup page
  And I fill in 'username' with 'padillac'
  And I fill in 'password' with 'secret'
  And I fill in 'password confirmation' with 'secret'
  And I fill in 'first name' with 'Pat'
  And I fill in 'last name' with 'Maddox'
  And I fill in 'email' with 'pat.maddox@gmail.com'
  And I select 'male' as gender
  When I sign up
  Then I should see 'Welcome, padillac'
```

Declarative

```
Scenario: Customer registers for the first time  
  Given I am at the new user signup page  
  When I sign up with username 'padillac'  
  Then I should see 'Welcome, padillac'
```

Which style to use?

- In general I prefer declarative. Less noise
- Imperative can be good for very sequential scenarios
- Try to capture the essence of the scenario
- **WHICHEVER THE CUSTOMER WANTS**

Semantic Elements

- Given
 - Used for setting up state, assumptions
- When
 - The action taking place
- Then
 - Expectations are verified against resulting state

```
Scenario: <Scenario title>  
  Given <some known state>  
  And <some more known state>  
  When <I do something>  
  Then <some desired outcome>  
  And <some desired outcome>
```

Focus on Business Value

- Pop the “why” stack
- Business value: how does this feature
 - make us money?
 - save us money?
 - protect valuable assets?

A user should enter his phone number when checking out.

Why?

So that we can call them to verify the order

Why?

So that we can minimize fraud

Why?

So that we don't lose money from fraudulent orders.

Feature Narrative

- Explains business value
- States actionable feature
- May include pertinent roles

Feature: <Title>

In order to <some business value>

<Actor> will need

to <some action>

Feature: Entering phone # upon checkout

In order to minimize fraud

Customers will need

to enter their phone numbers when checking out

Feature: <Title>

In order to <some business value>

As a <Role> I want <Actor>

to <some action>

Feature: Entering phone # upon checkout

In order to minimize fraud

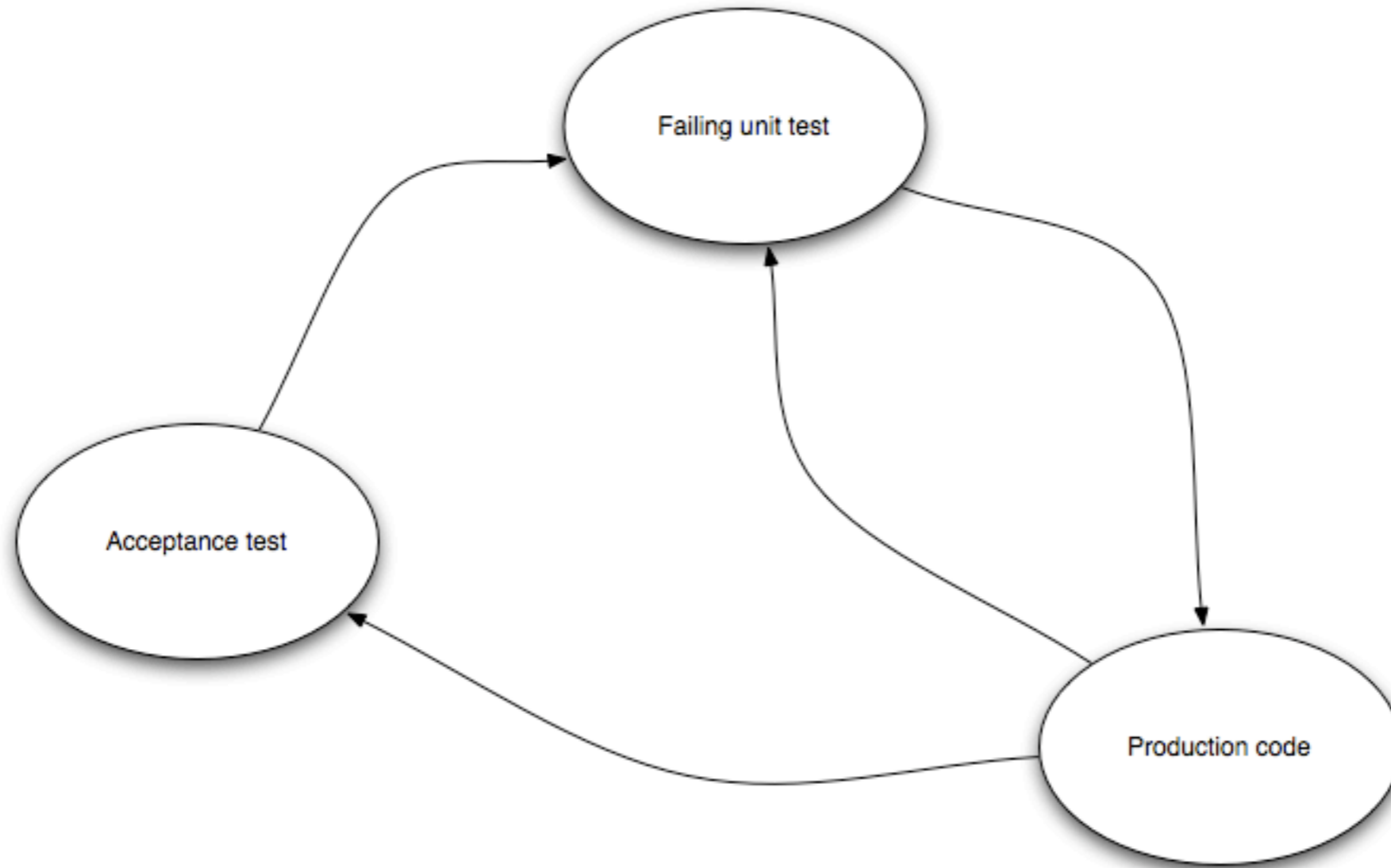
As a web sales rep, I want Customers

to enter their phone numbers when checking out

Exercise: Writing features for Blackjack

- When player wins he should be paid 1-1
- When player loses, he is not paid
- When player has blackjack, he is paid 1.5-1

The full BDD workflow



Outside-In

- Begin with outer edges of app
- Write examples to drive design, mocking dependencies in the layer below
- When you finish the outer layer, spec out the supporting objects

Exercise: Implementing a feature Outside-In

- `git co -b cucumber origin/cucumber`
- `git merge just_specs #` (or `origin/aces_solution` if you did not finish)
- `cucumber features/play_game.feature`

Addison-Wesley Professional Ruby Series



??? BDD ??? RSPEC

BY PAT MADDOX

Resources

- <http://rspec.info>
- <http://github.com/aslakhellesoy/cucumber/wikis>
- [http://evang.eli.st/blog](http://evang eli.st/blog) - My Awesome Blog