

# RSpec patterns and shmatterns

# RSpec ~~patterns~~ and shmatterns

Patterns are boring. Shmatterns are fun.

```
describe "Test-driven development" do  
  it { should be_fun }  
end
```

# Setting expectations up

- What matters (or, “what do I test?”)

# Setting expectations up

- What matters (or, “what do I test?”)
- Cases 80% of applications have

# Setting expectations up

- What matters (or, “what do I test?”)
- Cases 80% of applications have
- Understanding example groups

# Setting expectations up

- What matters (or, “what do I test?”)
- Cases 80% of applications have
- Understanding example groups
- Spec outline

# Setting expectations up

- What matters (or, “what do I test?”)
- Cases 80% of applications have
- Understanding example groups
- Spec outline
- Shared example groups



# Setting expectations up

- What matters (or, “what do I test?”)
- Cases 80% of applications have
- Understanding example groups
- Spec outline
- Shared example groups
- How SEGs correspond to Ruby modules on implementation side

# Setting expectations up

- What matters (or, “what do I test?”)
- Cases 80% of applications have
- Understanding example groups
- Spec outline
- Shared example groups
- How SEGs correspond to Ruby modules on implementation side
- 72 slides

# Before we start

- `git clone git://github.com/michaelklishin/ruby_barcamp_kiev_nov_2009.git`

# What matters

- Public API

# What matters

- Public API
- Something object exposes to other objects

# What matters

- Public API
- Something object exposes to other objects
- *Communication between objects* matters

# What matters

- Public API
- Something object **exposes to other** objects
- **Communication between objects** matters
- How exactly work gets done is **secondary**

# What matters

- What **role** object plays



# What matters

- What role object plays
- Observer

# What matters

- What role object plays
- Observer
- Observable

# What matters

- What role object plays
- Observer
- Observable
- SubscriptionTopic

# What matters

- What role object plays
- Observer
- Observable
- SubscriptionTopic
- TopicSubscriber

# What matters

- What object responds to

# What matters

- What object responds to
- Edge cases & contexts

# What matters

- What object responds to
- Edge cases & contexts
- **Language:** RSpec was created for less tech savvy people

# Example: messages object responds to

```
describe "Twitter account" do
  it { should respond_to(:sign_in) }

  it { should respond_to(:tweet) }

  it { should respond_to(:subscribe) }

  it { should respond_to(:unsubscribe) }

  it { should respond_to(:send_direct_message) }
end
```



# Example: contexts

```
describe "Basketball lover" do
  context "in the summer" do
    it "plays outdoors"
  end

  context "in the winter" do
    it "plays indoors"
  end
end
```

# Example: poor language

```
describe "Blog post" do  
  it "is validating title presence"  
end
```

Here we see how  
language used in the spec  
does not really explain  
what happens and why

# Example: better language

```
describe "A blog post" do
  context "without title" do
    it "IS NOT valid"
  end

  context "with a title 25 characters long" do
    it "is valid"
  end
end
```

# 80%

- Valid vs invalid
- Public vs private
- Entitled (with title)
- Textual body
- Authored (things that have author)
- Groups of people (projects, communities, et cetera)
- Subscribe/unsubscribe

# Valid vs invalid: what matters

- What **states** object can be in
- In what states it is **valid**
- In what states it is **NOT valid**
- What happens when **state changes**

# Example: valid vs invalid, spec outline

```
describe "A blog post" do
  context "without title" do
    it "IS NOT valid"
  end

  context "with a title 0 characters long" do
    it "is NOT valid"
  end

  context "with a title 25 characters long" do
    it "is valid"
  end

  context "with a title 37000 characters long" do
    it "IS NOT valid"
  end
end
```

# Spec outline

- Example group with **contexts**
- Contexts have **example titles**
- Examples have no bodies
- Used to **communicate your ideas** to other people
- A great lightweight replacement to the ugly bloated monster that UML is
- Is manifold times more useful if you use Emacs + GitHub Gist + Campfire + Skype

# Spec outline helps...

- Understand **what is unclear** in requirements
- Identify **contexts**
- Reveal **edge cases**
- Improve **API** design
- Improve **language in specs & implementation** (module names, method names, etc)



# Example: capturing similarities

```
describe "A project" do
  context "when just started" do
    it "has only one membership"

    it "has only one member"

    it "has one administrator"
  end

  context "when one more person is added" do
    it "has two membership"

    it "has two members"

    it "still has one administrator"

    it "still has one regular member"
  end
end
```

# Example: capturing similarities

```
describe "A community" do
  context "when just started" do
    it "has only one membership"

    it "has only one member"

    it "has one administrator"
  end

  context "when one more person is added" do
    it "has two membership"

    it "has two members"

    it "still has one administrator"

    it "still has one regular member"
  end
end
```

```
shared_examples_for "Group of people" do
  context "when just started" do
    it "has only one membership"

    it "has only one member"

    it "has one administrator"
  end

  context "when one more person is added" do
    it "has two membership"

    it "has two members"

    it "still has one administrator"

    it "still has one regular member"
  end
end

describe "A community" do
  it_should_behave_like "Group of people"
end

describe "A project" do
  it_should_behave_like "Group of people"
end
```

# Shared example groups (SEGs) help

- Remove **duplication in specs**
- Identify **reusable pieces of behavior** in implementation (hint: **Ruby modules**)

# Full SEG example

- See in Git repository at `spec/  
full_group_of_people_example_seg.rb`

# How to deal with shared state

- Step one: @model
- See spec/  
group\_of\_people\_with\_ivar\_for\_state\_sharing\_example\_seg.rb

# How to deal with shared state

- Step one: @model
- Step two: #model method

# How to deal with shared state

- Step one: @model
- Step two: #model method
- Step three: RSpec's built-in #subject



# How to deal with shared state

- Step one: @model
- Step two: #model method
- Step three: RSpec's built-in #subject
- Step four: #subject can even be omitted

```
shared_examples_for "Group of people" do
  context "when just started" do
    it "has only one membership" do
      # use #count or have(1).memberships here,
      # it is not the point at all
      @model.memberships.size.should == 1
    end

    it "has only one member" do
      @model.members.size.should == 1
    end

    it "has one administrator" do
      @model.administrators.size.should == 1
    end

    it "has NO regular members" do
      @model.regular_members.should be_empty
    end
  end
end
end
```

```
describe "A community" do
  before :each do
    @model = Community.new
  end

  it_should_behave_like "Group of people"
end

describe "A project" do
  before :each do
    @model = Project.new
  end

  it_should_behave_like "Group of people"
end
```

# Common API

- :memberships
- :members
- :membership\_of(person)
- :member?(person)
- :add\_member(person, options = {})
- :remove\_member(person)
- :administrators
- :regular\_members

```
module Traits
  # Provides common functionality for groups of people
  module GroupOfPeople

    #
    # API
    #

    def members
    end # members

    def memberships
    end # memberships

    def administrators
    end # administrators

    def regular_members
    end # regular_members

    def add_member(person, options = {})
    end # add_member(person, options = {})

    def remove_member(person)
    end # remove_member(person)
  end # GroupOfPeople
end # Traits
```

# Modules are just **SEG counterparts** in **implementation land**

- “group of people” => `Traits::GroupOfPeople`

# Publish/Subscribe at GitHub

- You can subscribe to people
- You can subscribe to repositories

# Publish/subscribe: contexts

- Initially
- Before subscribing
- After subscribing
- After subscribing and then unsubscribing
- When subscribing to same topic twice
- When unsubscribing from same topic twice



```
describe "Repository" do
  it_should_behave_like "Traits::PubSub::Topic"
end
```

```
describe "Person" do
  it_should_behave_like "Traits::PubSub::Topic"
end
```

```
module Traits
  module SubscriptionTopic

    #
    # Public API
    #

    def subscribe(person)
    end # subscribe(person)

    def unsubscribe(person)
    end # unsubscribe(person)

    def subscription_of(person)
    end # subscription_of(person)

    def subscribed?(person)
    end # subscribed?(person)

    # ...
  end # SubscriptionTopic
end # Traits
```

# Public vs private: contexts

- Initially
- Before publishing
- After publishing
- After publishing and then unpublishing

# Public vs private: trait (module)

- Traits::Publishable
- :publish
- :unpublish
- :published?
- :public?
- :private?

# Workflow

- Write spec outline

# Workflow

- Write **spec outline**
- Show it to **someone else**

# Workflow

- Write **spec outline**
- Show it to **someone else**
- **Correct** language & behavior

# Workflow

- Write **spec outline**
- Show it to **someone else**
- **Correct** language & behavior
- **Write specs** one example at a time



# Workflow

- Write **spec outline**
- Show it to **someone else**
- **Correct** language & behavior
- **Write specs** one example at a time
- **Write implementation** to make 'em pass

# Workflow

- Write **spec outline**
- Show it to **someone else**
- **Correct** language & behavior
- **Write specs** one example at a time
- **Write implementation** to make 'em pass
- Extract **SEGs**

# Workflow

- Write **spec outline**
- Show it to **someone else**
- **Correct** language & behavior
- **Write specs** one example at a time
- **Write implementation** to make 'em pass
- Extract **SEGs**
- Extract **traits/modules**

# Workflow

- Write **spec outline**
- Show it to **someone else**
- **Correct** language & behavior
- **Write specs** one example at a time
- **Write implementation** to make 'em pass
- Extract **SEGs**
- Extract **traits/modules**
- Group SEGs & traits into **libraries**

# Some observations

- Works well with people long ways away from you

# Some observations

- Works well with people long ways away from you
- Does not work with **lazy bastards**

# Some observations

- Works well with people long ways away from you
- Does not work with **lazy bastards**
- Requires certain level of discipline (like TDD itself)

# Some observations

- Works well with people long ways away from you
- Does not work with **lazy bastards**
- Requires certain level of discipline (like TDD itself)
- Leads to **perspective shift** fairly quickly



# Caveats

- Analysis paralysis

# Caveats

- Analysis paralysis
- “Over-generalization”

# Caveats

- Analysis paralysis
- “Over-generalization”
- It is hard to pick **great names**

# Caveats

- Analysis paralysis
- “Over-generalization”
- It is hard to pick **great names**
- Never try to generalize before you see **real similarities or duplication**

# A minimum to get started

- Start using spec outlines

# A minimum to get started

- Start using spec outlines
- Show them to other people

# A minimum to get started

- Start using spec outlines
- Show them to other people
- Attach them to tickets so others can see them

# I've told you

RSpec shmatterns are fun



# Thank you