# PDX.rb Beginners' Meetup
# Testing 101

Sam Livingston-Gray

# TL;DR

- This presentation is too long!
  *(Like Pascal, I lacked the time to make it shorter.)*
- In order to not clobber the discussions and peer support here, I will **STOP TALKING** after 30 minutes!
- Please ***ask questions*** if I go too fast!
  *It's more important for you to learn than for me to talk.*

# Outline

- Describe a Stack
- Talk briefly about the narrative structure of tests
  - ...with some example code
- Compare Test::Unit, RSpec, and Cucumber
- Q&A

# Data Structure: Stack

- All of my examples will describe a stack.
- According to Wikipedia,

> *"In computer science, a stack is a last in, first out (LIFO) abstract data type and linear data structure."*

- ...Clear as mud, right?



I'll be showing the same set of tests in a lot of different contexts.
So that you're not too distracted by the content, I want to take just a few minutes to talk about what a stack is.

# Stack: Definition

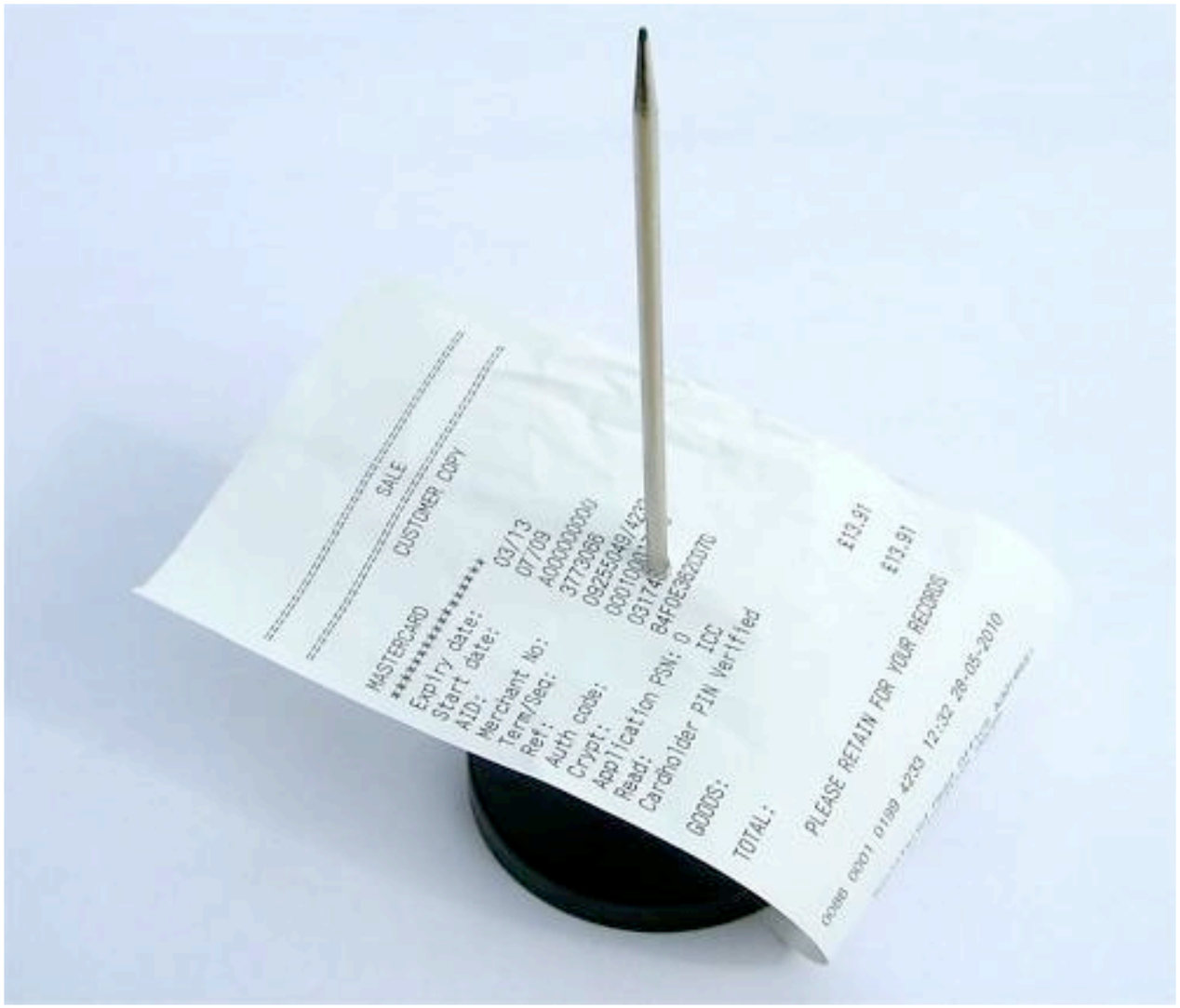
*"In computer science, a stack is a last in, first out (LIFO) abstract data type and linear data structure."*

Source: Wikipedia

...Clear as mud, right?

# Stack: Examples

```
class Stack
    def initialize
        @array = []
    end

    def empty?
        @array.empty?
    end

    def size
        @array.size
    end

    def peek
        @array.first
    end

    def push(thing)
        @array.unshift thing
    end

    def pop
        @array.shift
    end
end
```


SALE
CUSTOMER COPY
MASTERCARD
Expiry date:
Start date:
AID:
Merchant No:
Term/Seq:
Ref:
Auth code:
Crypt:
Application PSN: 0
Read:
Cardholder PIN Verified
ICC
GOODS:
£13.91
TOTAL:
£13.91
PLEASE RETAIN FOR YOUR RECORDS

# Stack: Behavior

- A stack keeps track of a list of things.
- There are three things we can do with a stack:
  - **Peek:** See what the thing on top of the stack is.
  - **Push:** Put one new thing onto the top of the stack.
  - **Pop:** Take one thing off of the top of the stack.

# Stack: A Story

The following cautionary tale is swiped from Seth Godin's blog...

# Stack: A Story

*"I want to wax the car today."*

**TO DO:**

- **Wax car**

# Stack: A Story

*"Oops, the hose is still broken from the winter. I'll need to buy a new one at Home Depot."*

**TO DO:**

- **Get hose at Home Depot**
- Wax car

# Stack: A Story

*"But Home Depot is on the other side of the Tappan Zee bridge and getting there without my EZPass is miserable because of the tolls."*

*"But, wait! I could borrow my neighbor's EZPass..."*

**TO DO:**

- **Borrow neighbor's EZPass**
- Get hose at Home Depot
- Wax car

# Stack: A Story

*"Bob won't lend me his EZPass until I return the mooshi pillow my son borrowed, though."*

**TO DO:**

- **Return mooshi pillow**
- Borrow neighbor's EZPass
- Get hose at Home Depot
- Wax car

# Stack: A Story

*"And we haven't returned it because some of the stuffing fell out and we need to get some yak hair to restuff it."*

**TO DO:**

- **Acquire yak hair**
- Return mooshi pillow
- Borrow neighbor's EZPass
- Get hose at Home Depot
- Wax car

# Stack: A Story

*"And the next thing you know,*

*you're at the zoo,*

*shaving a yak,*

*all so you can wax your car."*

# Stack: Review

To make sure everyone understood a stack:

- I gave a **definition**.
- I showed some **examples**.
    1. Some Ruby code
    2. IKEA stacking rings
    3. A receipt spike
- I listed its **behaviors**.
- I told a **story** that illustrated how a stack behaves.



There is a point to this, I promise...

# Code vs. Tests

Code is like that Wikipedia **definition**:

- precise,
- technical,
- pithy.

All the ***repetition*** has been carefully squeezed out of it.

Tests give **examples** that describe the **behavior** of your code.

Each test should tell one small **story**.

Repetition in tests can be useful, if it helps the reader follow the story.

# Telling the Story

- No matter what kind of tests you're writing,
- or which testing framework you use,
- every test should have a beginning, a middle, and an end.

# Telling the Story

From http://c2.com/cgi/wiki?ArrangeActAssert

- **Arrange** all necessary preconditions and inputs.
- **Act** on the object or method under test.
- **Assert** that the expected results have occurred.

# Telling the Story

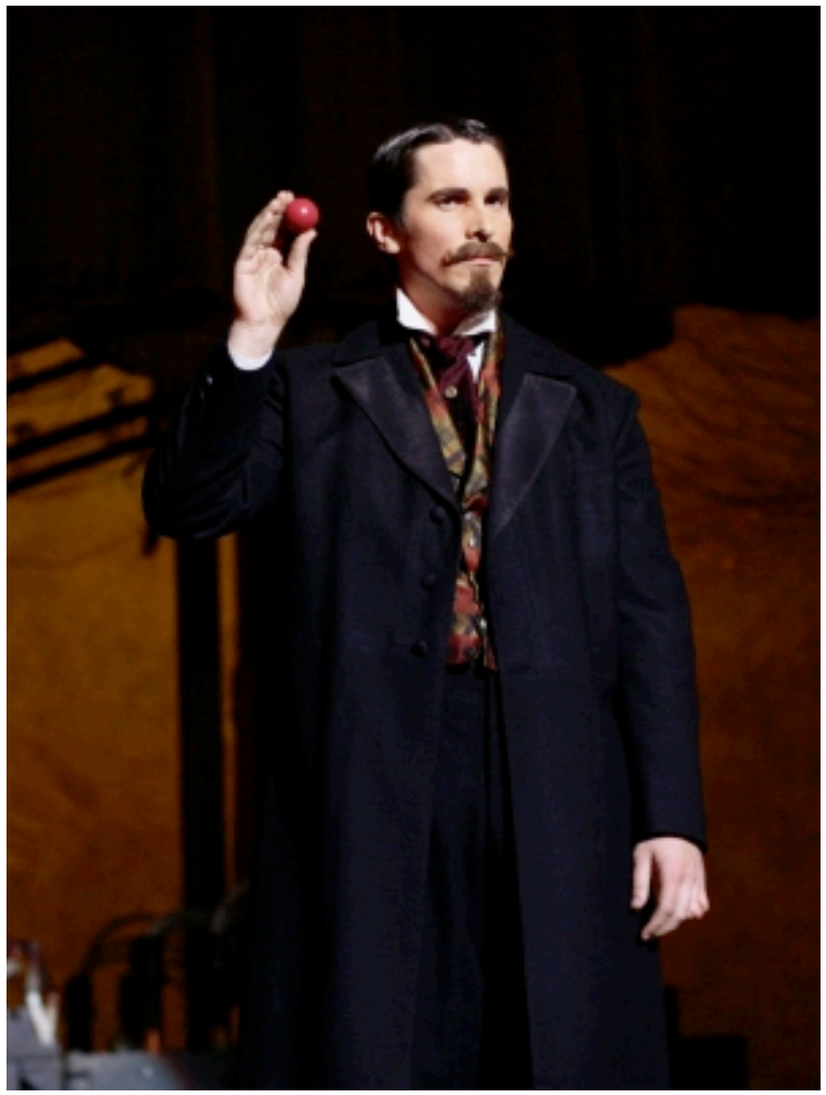
I also see this referred to as Given/When/Then:

- **Given** (some background),
- **When** I take some action,
- **Then** I expect to see certain results.

# Telling the Story

Or, in *(not-very-good)* movies about stage magic:

- **The Pledge**
- **The Turn**
- **The Prestige**



# Code or
# It Didn't Happen



I'm going to show some

# Code or It Didn't Happen

- I'm going to show a set of tests that describe a Stack object.
- We'll start with no framework whatsoever, and add in a few little features.
- After that, we'll look at the equivalent tests in three popular Ruby testing frameworks.

# Plain old Ruby code

# Plain old Ruby code

Let's start by describing our Stack class using plain old Ruby code—no testing framework of any kind.

These tests just use comments and whitespace to tell their story, so we can focus on identifying the parts of the story.

`0_no_testing_framework.rb`

# Plain old Ruby code

*(one test)*

```
##### A stack with one thing on it... #####

# is not empty / has size of 1
stack = Stack.new
stack.push :foo
raise "Stack with one item is empty" if stack.empty?
raise "Stack with one item doesn't have size == 1"
  if stack.size != 1
```

# Plain old Ruby code

*(one test)*

```
##### A stack with one thing on it... #####

# is not empty / has size of 1
stack = Stack.new
stack.push :foo
raise "Stack with one item is empty" if stack.empty?
raise "Stack with one item doesn't have size == 1"
  if stack.size != 1
```

Arrange | Act | Assert



**Arrange** all necessary preconditions and inputs.
**Act** on the object or method under test.
**Assert** that the expected results have occurred.

# Plain old Ruby code

*(one assertion)*

```
raise "Stack with one item is empty" if stack.empty?
```

...is equivalent to...

```
if stack.empty?
  raise "Stack with one item is empty"
end
```



In other words: “If the following statement is true, STOP THE ENTIRE PROGRAM RIGHT HERE.”

# def assert

# def assert

Let's extract that "raise complaint if something went wrong" pattern:

```
def assert(truth, failure_message)
  raise failure_message unless truth
end
```

1_worlds_smallest_testing_framework.rb



That "raise complaint if something went wrong" pattern makes it a little hard to see what's going on, so let's extract that.
Notice also that we've switched from "if" to "unless". Our brains aren't very good at negation, so it's often clearer to assert what you DO want, instead of refuting what you DON'T want.

# def assert

*(one test)*

```
##### Stack with one thing on it #####

stack = Stack.new
stack.push :foo
assert !stack.empty?, "Stack with one item is empty"
assert stack.size == 1,
  "Stack with one item doesn't have size == 1"
```

# def assert

*(one test)*

```
##### Stack with one thing on it #####

stack = Stack.new
stack.push :foo
assert !stack.empty?, "Stack with one item is empty"
assert stack.size == 1,
  "Stack with one item doesn't have size == 1"
```

Arrange | Act | Assert



The only thing that's changed here is the assertion. We'll compare the two directly on the next slide...

# def assert

*(one assertion)*

```
raise "Stack with one item is empty" unless stack.empty?
```

...is equivalent to...

```
assert stack.empty?, "Stack with one item is empty"
```



Admittedly, this isn't a huge change. But it does put the assertion first, which the other syntax didn't let us do.
I took this a bit further, but (a) this presentation is too long already, and (b) I realized I was starting to reimplement RSpec. So...

# Test::Unit, RSpec, and Cucumber



This is the part where I compare what I think are the Big Three testing frameworks for Ruby. They're not the only three, but most of the others are relatively fringy, and all of them that I'm aware of tend to look a lot like either Test::Unit or RSpec.

# Test::Unit

# Test::Unit

- Part of Ruby standard library
  - *(at least, in Ruby 1.8. Ruby 1.9 uses MiniTest, with Test::Unit changed to a thin compatibility layer)*
- Tests are grouped together in subclasses of Test::Unit::TestCase
- Tests are in methods named test_*

# Test::Unit

- Each test is run against a new TestCase object; results are summarized to screen
- State common to all tests can be managed in the #setup and #teardown hook methods
- Basic set of assertions (assert_equal, assert_nil, assert_raise, &c); easy to define others

# Test::Unit

- See: comparison/stack_test.rb
- Three separate classes
- Test names can be hard to read
  - ActiveSupport::TestCase adds a class macro that lets you use strings to describe your tests

# Test::Unit

- Arrange: in #setup
- Act/Assert: in test_* methods

# RSpec

# RSpec

- Have to install via Rubygems
- Uses an "internal DSL" for organizing ~~tests~~ specs
  - including Kernel#should, yuck (2.11 introduces alternate syntax)
- Has quite a few advanced features: shared examples, .let, implicit subject, and more

# RSpec

- Groups ~~tests~~ specs in a “describe block”
- Individual ~~tests~~ specs are defined in an “it block”.
- Uses “matchers” instead of assertions; defining custom matchers isn’t bad once you RTFM

# RSpec

- See: comparison/stack_spec.rb
- Note nested .describe blocks
- Note use of .subject method
  - it { should be_empty } # *implicit*
  - expect(stack.peek).to be_nil # *explicit*

# RSpec

- Arrange:
  in .subject, .let, .before(:all), .before(:each)
- Act/Assert: in .it blocks

# Cucumber

# Cucumber

- Easily the most complicated of the three
- Have to install via Rubygems
- Uses an "external DSL" called Gherkin for writing ~~tests~~ ~~specs~~ scenarios
- Scenarios consist of steps; "step definitions" are implemented in Ruby

# Cucumber

- ~~Tests~~ ~~Specs~~ Scenarios are grouped into Features
- ~~Tests~~ ~~Specs~~ Scenarios start with "Scenario:" followed by a description
- Shared setup goes in a "Before:" block
- Steps start with the words "Given", "When," and "Then"

# Cucumber

- See: comparison/stack.feature
- You can write these steps to say anything you want, at whatever level of detail you want

- I'd probably combine "the stack (is/is not) empty" with "the stack has size N"
- I didn't implement step definitions for this (ran out of time)

# Comparison

# Test::Unit

- Pluses:
  - No external dependencies
  - Familiar to users of xUnit tools from other languages
- Minuses:
  - Need to “roll your own” if you want extra features

# RSpec

- Pluses:
  - More expressive than Test::Unit
  - Very popular
- Minuses:
  - Magical, bit of a learning curve
  - Kernel#should is evil (see also: magical)

# Cucumber

- Pluses:
  - *Extremely* expressive
  - Accessible to non-developers
- Minuses:
  - Many more moving parts just to execute one line of Cucumber
  - Typical stack: Cucumber+Capybara+Selenium +Browser+Rails = sloooow

# Made it!

(The end)