

Week 6

your requests

RSpec

Model a T-Shirt

```
class Shirt
  attr_accessor :color
  def initialize(color)
    @color = color
  end
end

class TShirt
  attr_accessor :logo
  def initialize(color, logo)
    super(color)
    @logo = logo
  end

  def ironic?
    @logo == 'Hall and Oates'
  end
end
```

RSpec Test

```
require 'shirt'

describe Shirt do
  before(:all) do
    # ...
  end

  it "should remember what color it is" do
    shirt = Shirt.new('blue')
    shirt.color.should == 'blue'
  end
end
```

spec test_file.rb

```
$ spec spec_shirt.rb
```

```
.
```

```
Finished in 0.007577 seconds
```

```
1 example, 0 failures
```

Test for Sleeves

```
require 'shirt'

describe Shirt do
  before(:all) do
    # ...
  end

  it "should remember what color it is" do
    shirt = Shirt.new('blue')
    shirt.color.should == 'blue'
  end

  it "should have a number of sleeves" do
    shirt = Shirt.new('blue')
    shirt.sleeves.should == 2
  end
end
```

Add some sleeves

```
$ spec spec_shirt.rb
```

```
.F
```

```
1)
```

```
NoMethodError in 'Shirt should have a number of sleeves'  
undefined method `sleeves' for #<Shirt:0x5b8dd0 @color="blue">  
./spec_shirt.rb:15:
```

```
Finished in 0.007913 seconds
```

```
2 examples, 1 failure
```

Pop your collar



COOLNESS

You may be cool...but you'll never be 4 popped-collars cool.



Try to Pop Collar

```
require 'shirt'

describe Shirt do
  before(:each) do
    @shirt = Shirt.new('blue')
  end

  it "should remember what color it is" do
    @shirt.color.should == 'blue'
  end

  it "should have a number of sleeves" do
    @shirt.sleeves.should == 2
  end

  it "should raise when trying to pop a collar" do
    lambda { @shirt.pop_collar! }.should raise_error(RuntimeError)
  end
end
```

Mocking and Stubbing

Test/Unit Random

```
class Foo
  def random_method
    rand(6)
  end
end
```

```
class TestThatStubbingWorks < Test::Unit::TestCase
  def test_custom_random_method
    foo = Foo.new
    assert_equal(5, foo.random_method)
  end
end
```

RSpec Random

```
class Foo
  def random_method
    rand(6)
  end
end

describe Foo do
  it "should return a random number" do
    foo = Foo.new
    foo.random_method.should == 5
  end
end
```

How do we test this?

Random Flexmock

```
require 'rubygems'
require 'flexmock/test_unit'
require 'test/unit'

class Foo
  def random_method
    rand(6)
  end
end

class TestThatStubbingWorks < Test::Unit::TestCase
  def test_custom_random_method
    foo = Foo.new
    flexmock(foo).should_receive(:random_method).and_return(5)
    assert_equal(5, foo.random_method)
  end
end
```

Random Mocha

```
require 'test/unit'  
require 'rubygems'  
require 'mocha'
```

```
class Foo  
  def random_method  
    rand(6)  
  end  
end
```

```
class TestThatStubbingWorks < Test::Unit::TestCase  
  def test_custom_random_method  
    foo = Foo.new  
    foo.expects(:random_method).returns(5)  
    assert_equal(5, foo.random_method)  
  end  
end
```

Random RSpec

```
class Foo
  def random_method
    rand(6)
  end
end

describe Foo do
  it "should return a random number" do
    foo = Foo.new
    foo.stub!(:random_method).and_return(5)
    foo.random_method.should == 5
  end
end
```

What if our object was
passed around?

```
class TestSlotMachine < Test::Unit::TestCase
  def test_slot_machine_returns_3_numbers
    number_generator = NumberGenerator.new(10)
    slot_machine = SlotMachine.new(number_generator)

    result = slot_machine.pull_handle
    assert_equal(3, result.length)
    assert(result.all? { |pick| (0..10) === pick })
  end
end
```

What do we need to
test?

flexmock

```
class TestSlotMachine < Test::Unit::TestCase
  def test_slot_machine_uses_number_generator
    number_generator = NumberGenerator.new(10)
    slot_machine = SlotMachine.new(number_generator)

    flexmock(number_generator).should_receive(:random_number).times(3)

    slot_machine.pull_handle
  end
end
```

with()

```
class TestSlotMachine < Test::Unit::TestCase
  def test_number_generator_uses_rand
    number_generator = NumberGenerator.new(10)
    slot_machine = SlotMachine.new(number_generator)

    flexmock(number_generator).should_receive(:rand).with(10).times(3)

    slot_machine.pull_handle
  end
end
```

mocha

```
class TestSlotMachine < Test::Unit::TestCase
  def test_slot_machine_uses_number_generator
    number_generator = NumberGenerator.new(10)
    slot_machine = SlotMachine.new(number_generator)

    number_generator.expects(:random_number).times(3)

    slot_machine.pull_handle
  end
end
```

```
class TestSlotMachine < Test::Unit::TestCase
  def test_number_generator_uses_rand
    number_generator = NumberGenerator.new(10)
    slot_machine = SlotMachine.new(number_generator)

    number_generator.expects(:rand).with(10).times(3)

    slot_machine.pull_handle
  end
end
```

rspec

Make sure we have 3 numbers

```
describe SlotMachine do
  it "should return 3 numbers" do
    number_generator = NumberGenerator.new(10)
    slot_machine = SlotMachine.new(number_generator)

    result = slot_machine.pull_handle
    result.length.should == 3
    result.each { |pick| (0..10).should == pick }
  end
end
```

```
describe SlotMachine do
  it "should use the random number generator" do
    number_generator = NumberGenerator.new(10)
    slot_machine = SlotMachine.new(number_generator)

    number_generator.should_receive(:random_number).exactly(3).times

    slot_machine.pull_handle
  end
end
```

```
describe SlotMachine do
  it "should use rand to generate a random number" do
    number_generator = NumberGenerator.new(10)
    slot_machine = SlotMachine.new(number_generator)

    number_generator.should_receive(:rand).exactly(3).times

    slot_machine.pull_handle
  end
end
```

Testing Strategies

TDD or BDD

It doesn't matter

Just write tests!

Unit Testing

Writing To The Database

When?

- Only when you must
- **NOT** for seed data
- But you **SHOULD** sometime

Why write?

- Validations must match database
- Your table may change
- Foreign keys

DO NOT

- Write to the db as a fixture substitute
- Access the network
- Mock too much
- Use code coverage to be definitive
- Write fragile tests

Functional Testing

Each Test Should

- Test **ONE** action
- **Not** test every parameter
- Use appropriate request method
- Verify instance variables
- Validate content

Integration Testing

Integration Tests

- Span multiple controllers/actions
- Verify how a user interacts
- Validate content

When To Write Tests

- Before implementation
- During implementation
- After implementation
- **ESPECIALLY** to reproduce bugs

When Testing Goes Wrong

Brittle Tests

```
class TestBrittle < Test::Unit::TestCase
  def test_too_brittle
    obj = AwesomeObject.new
    assert_equal(
      500,
      obj.some_retarded_business_rule_that_returns_a_number
    )
  end
end
```

NOT DRY

```
class TestNotDry < Test::Unit::TestCase
  def test_my_model_is_awesome
    model = MyModel.new(
      :name => 'Aaron Patterson',
      :address => '123 Awesome St',
      :phone => '228-1736'
    )
    assert(model.does_something)
  end

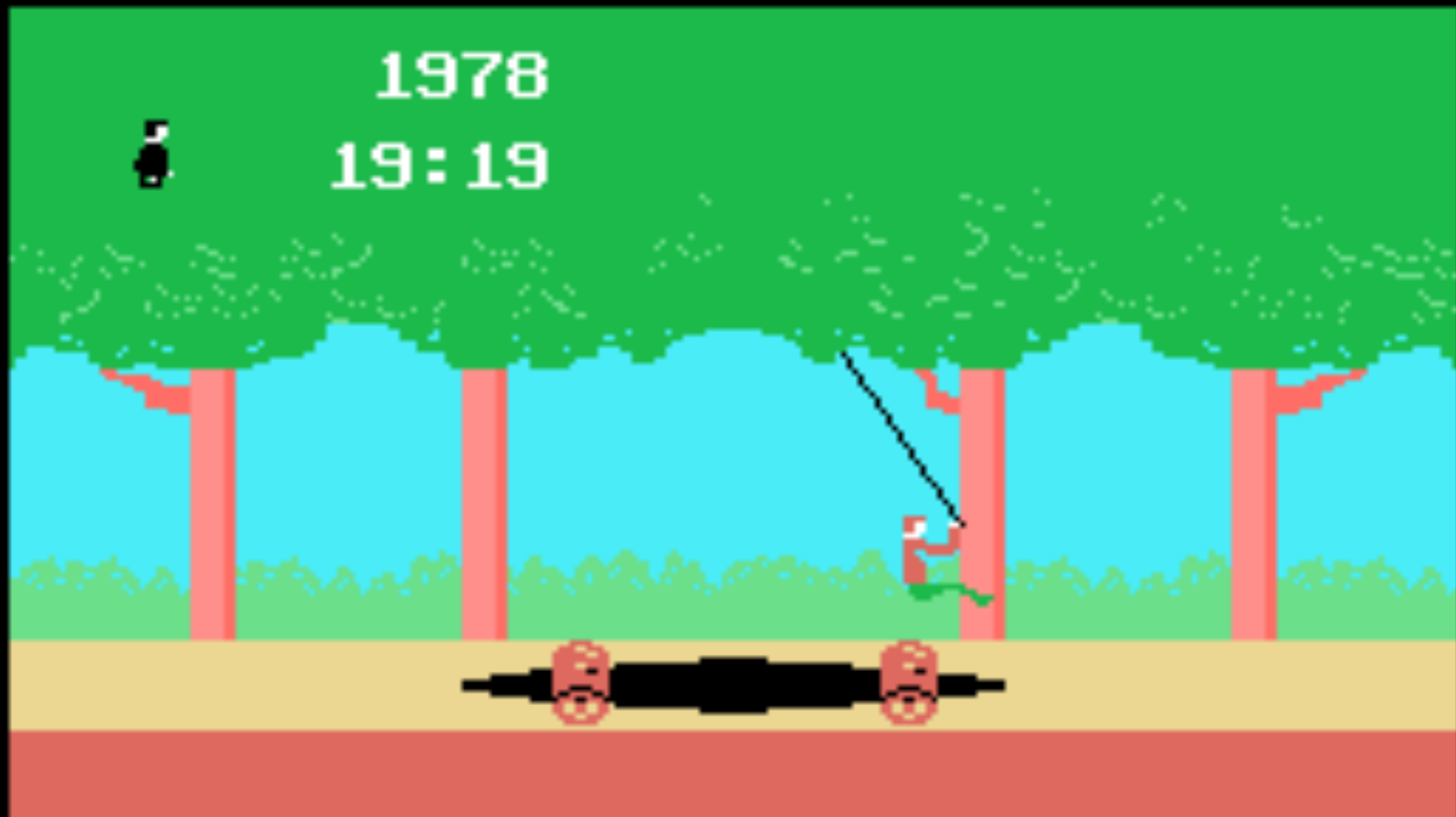
  def test_my_model_is_still_awesome
    model = MyModel.new(
      :name => 'Aaron Patterson',
      :address => '123 Awesome St',
      :phone => '228-1736'
    )
    assert(model.does_something_else)
  end
end
```

There are no tests!

Avoiding Pitfalls

1978

19:19



 ACTIVISION

Avoid the Noid

- Aggressively refactor
- Take the time to Do It Right
- Continuously Integrate
- Check code coverage