

Stanko Krtalić Rusendić
Ruby on Rails engineer @ Infinum

—

stanko.krtalic@gmail.com
[@monorkin](#)
<https://github.com/Stankec>

STUBS | MOCKS | SPIES

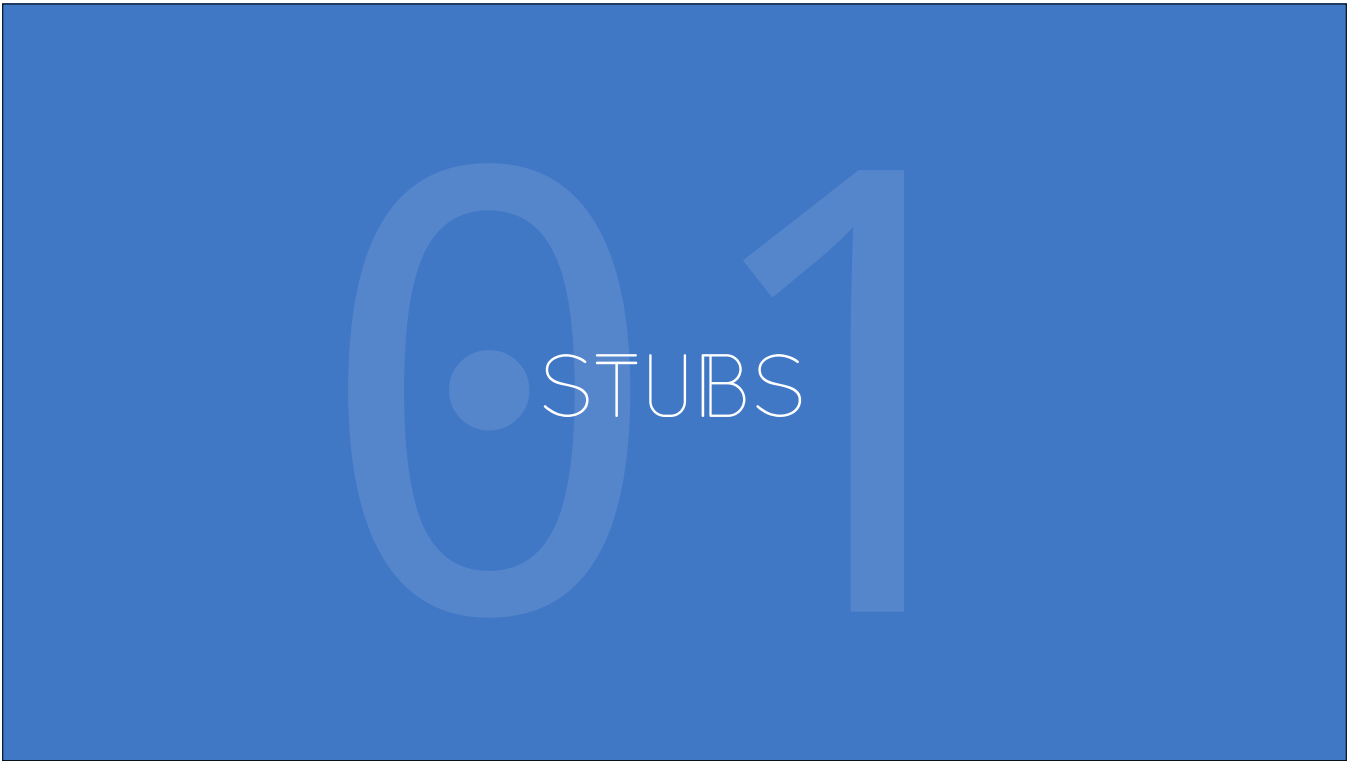
Hi everybody!

I will be talking about stubs, mocks and spies.

Before we begin, a quick show of hands.

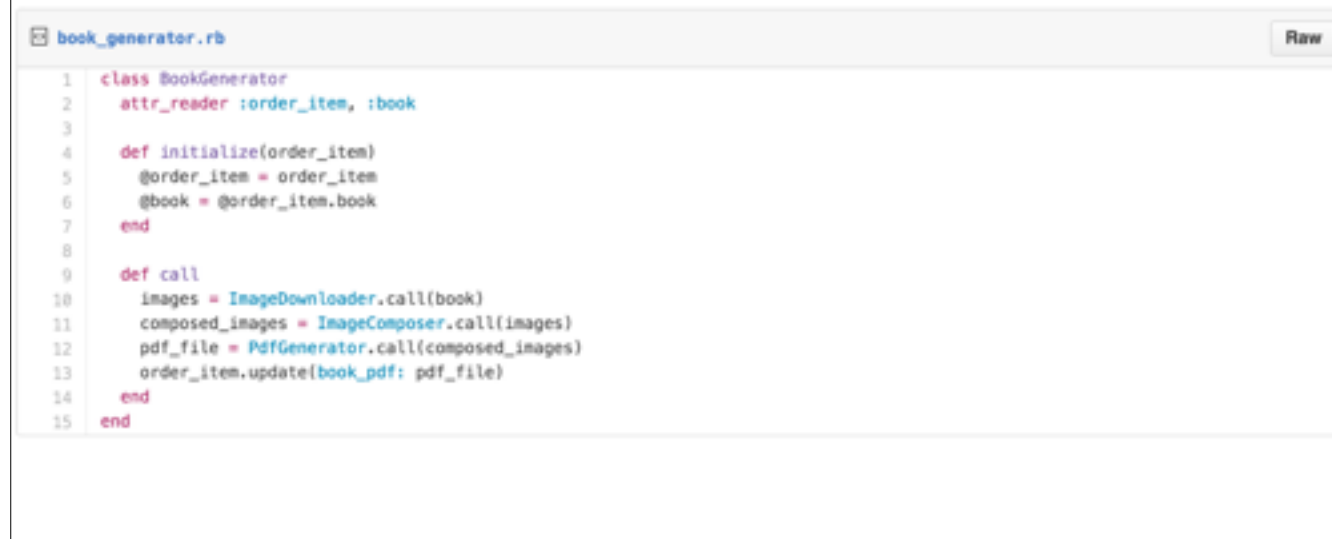
Who has used or knows what these are?

Stubs, mocks and spies are an integral part of every test suite. It's really complicated to test code without using at least one of these paradigms. I'm sure that at least some of you will say "Oh, I don't use them and my test suite works fine." and you would be completely right but give me a chance to explain my self.



Let's start off with stubs.

STUBS



```
book_generator.rb
1 class BookGenerator
2   attr_reader :order_item, :book
3
4   def initialize(order_item)
5     @order_item = order_item
6     @book = @order_item.book
7   end
8
9   def call
10    images = ImageDownloader.call(book)
11    composed_images = ImageComposer.call(images)
12    pdf_file = PdfGenerator.call(composed_images)
13    order_item.update(book_pdf: pdf_file)
14  end
15 end
```

Here is a simple class that generates PDF files and attaches them to the given order_item. As you can see it's pretty small, DRY and separates concerns to different services. Let's see the spec for this class.

STUBS

```
book_generator_spec.rb
1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       order_item = OrderItem.create(type: 'Book', title: 'My big adventure', configuration: { ... })
5       BookGenerator.new(order_item).call
6
7       expect(order_item.book_pdf).not_to be_nil
8     end
9   end
10 end
```

This surely isn't the worlds best spec, but it certainly will test if we returned something.

STUBS



```
book_generator.rb
1 class BookGenerator
2   attr_reader :order_item, :book
3
4   def initialize(order_item)
5     @order_item = order_item
6     @book = @order_item.book
7   end
8
9   def call
10    images = ImageDownloader.call(book)
11    composed_images = ImageComposer.call(images)
12    pdf_file = PdfGenerator.call(composed_images)
13    order_item.update(book_pdf: pdf_file)
14  end
15 end
```

The problem with the previous spec is that it relies upon the implementation of the ImageDownloader, ImageComposer, PdfGenerator and OrderItem classes.

Why is this a problem even?

Well to put it simply. You are relying and testing the implementation of other classes.

STUBS

```
book_generator_spec.rb
1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       order_item = OrderItem.create(type: 'Book', title: 'My big adventure', configuration: { ... })
5       BookGenerator.new(order_item).call
6
7       expect(order_item.book_pdf).not_to be_nil
8     end
9   end
10 end
```

By running this test we actually tested the behaviour of OrderItem, ImageDownloader, ImageComposer, PdfGenerator and finally BookGenerator.

This by it's self isn't that big of a problem. But, this way your test is concerned not only about the logic under test, but about logic from other modules that is being called by the code under test.

You don't know if all these classes work? You don't even know if this class exist!

Let's see how we can improve upon this test.

STUBS

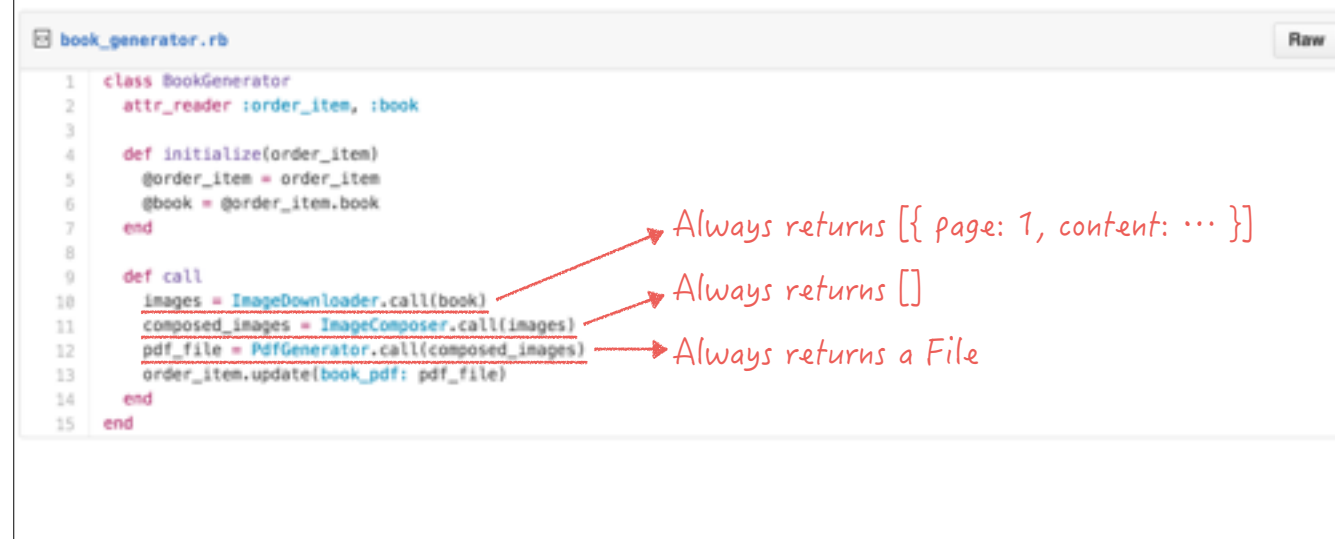
book_generator_spec.rb

Raw

```
1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       order_item = OrderItem.create(type: 'Book', title: 'My big adventure', configuration: { ... })
5
6       → allow(ImageDownloader).to receive(:call).and_return({ page: 1, images: [] })
7       → allow(ImageComposer).to receive(:call).and_return({})
8       → allow(PdfGenerator).to receive(:call).and_return(File.open('test_book.pdf', :r))
9
10      BookGenerator.new(order_item).call
11      expect(order_item.book_pdf).not_to be_nil
12    end
13  end
14 end
```

If you haven't noticed by now, all the examples here will be written in RSpec, but all major testing libraries implement the paradigmes I'm going to talk about. In RSpec we have the allow method. With it we can specify which method calls an object is allowed to receive and what it should return. Using it we can stub out the behaviour of objects.

STUBS



```
1 class BookGenerator
2   attr_reader :order_item, :book
3
4   def initialize(order_item)
5     @order_item = order_item
6     @book = @order_item.book
7   end
8
9   def call
10    images = ImageDownloader.call(book)
11    composed_images = ImageComposer.call(images)
12    pdf_file = PdfGenerator.call(composed_images)
13    order_item.update(book_pdf: pdf_file)
14  end
15 end
```

Always returns [{ page: 1, content: ... }]

Always returns []

Always returns a File

Now when we run the spec again it will give us the same result!

But how is this an improvement?

This time no external code got run! Well some did, if anybody noticed please hold your horses I will get to it in a moment.

So in this example. When we called ImageDownloader.call and passed it a book it didn't execute any logic it just returned the array we specified in the test.

We could completely delete the call method on the ImageDownloader class and this would still work!

This is a bit terrifying! But at the same time it's great! Using this we can test each object in isolation and control our program's flow.

The key feature being flow control!

```
articles_controller_spec.rb
1 describe ArticlesController, type: :controller do
2   describe '#create' do
3     context 'passed valid params' do
4       before do
5         → allow(Article).to receive(:create).and_return(true)
6       end
7
8       it 'redirects to the index page' do
9         ...
10      end
11    end
12
13    context 'passed invalid params' do
14      before do
15        → allow(Article).to receive(:create).and_return(false)
16      end
17
18      it 'renders the new template' do
19        ...
20      end
21    end
22  end
23 end
```

This is a really common situation.

You wrote a Rails controller and you want to test what it will do if you pass to it valid params and what will happen if you pass invalid params.

Usually people write two hashes in the beginning of the file, one containing a valid set of data and the other invalid data, and then they would get passed around in different tests. This is fine for simple CRUDs but can quickly get out of hand in more complex applications.

Stubs can help us with that problem! We can simply stub the call to the create method

STUBS



```
1 describe ArticlesController do
2   describe '#create' do
3     it "doesn't allow a second article to be created" do
4       article_1_params = { ... }
5       article_2_params = { ... }
6
7       allow(Article).to receive(:create).with(article_1_params).and_return(true)
8       allow(Article).to receive(:create).with(article_2_params).and_return(false)
9     end
10  end
11 end
```

Please note that this is a top level overview of stubs.

A really extensive set of methods and helpers exist to help you create exactly the flow you desire.

This example expects a certain parameter to be passed in a call for it to return true or false.

DOUBLES

book_generator_spec.rb

Raw

```
1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       → order_item = OrderItem.create(type: 'Book', title: 'My big adventure', configuration: { ... })
5
6       allow(ImageDownloader).to receive(:call).and_return({ page: 1, images: [] })
7       allow(ImageComposer).to receive(:call).and_return({})
8       allow(PdfGenerator).to receive(:call).and_return(File.open('test_book.pdf', :r))
9
10      BookGenerator.new(order_item).call
11      expect(order_item.book_pdf).not_to be_nil
12    end
13  end
14 end
```

Now that we have a basic understanding of what stubs are we can spot something that's not right here.

We still depend on an external module!

Not only that, but for our spec to pass we need a database, we need to read data from and write data to it.

This is unnecessary as we rely that our database, ActiveRecord and ActiveRecord all work.

Perhaps we can fix this issue with stubs?

DOUBLES

```
book_generator_spec.rb

1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       book = double(:book)
5       order_item = double(:order_item)
6
7       allow(order_item).to receive(:book).and_return(book)
8       allow(order_item).to receive(:update).and_return(book)
9
10      allow(ImageDownloader).to receive(:call).and_return({ page: 1, images: [] })
11      allow(ImageComposer).to receive(:call).and_return({})
12      allow(PdfGenerator).to receive(:call).and_return(File.open('test_book.pdf', :r))
13
14      BookGenerator.new(order_item).call
15      expect(order_item.book_pdf).not_to be_nil
16    end
17  end
18 end
```

In RSpec there exists a concept of doubles. You can think of them as dummy or empty objects. Each test library calls these objects differently so maybe this is familiar to you or you will have to do some research.

Here we create two empty objects and tell the `order_item` object what it should do when it receives a call to the book and update methods.

Now we don't depend on the implementation of OrderItem or Book!

As you can see we create empty objects with the `double` method there fore they are called doubles!

Something that confused me terribly when I had my first contact with doubles is the symbol in the brackets. What is that? Well that's the name of that particular object!

It's used only for debugging messages.



Now that we know what stubs are we can move on to mocks.

MOCKS

```
book_generator_spec.rb

1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       book = double(:book)
5       order_item = double(:order_item)
6
7       allow(order_item).to receive(:book).and_return(book)
8       allow(order_item).to receive(:update).and_return(book)
9
10      allow(ImageDownloader).to receive(:call).and_return({ page: 1, images: [] })
11      allow(ImageComposer).to receive(:call).and_return({})
12      allow(PdfGenerator).to receive(:call).and_return(File.open("test_book.pdf", :r))
13
14      BookGenerator.new(order_item).call
15      → expect(order_item.book_pdf).not_to be_nil
16    end
17  end
18 end
```

Perhaps you noticed in the last example that the spec now doesn't make sense.

And in fact won't pass.

This is because no `book_pdf` method is stubbed for the `order_item` double. And we shouldn't stub it because then we would be testing stubbed behaviour.

But we need to know that it got called because it's an integral part of our code.

MOCKS

```
book_generator_spec.rb
1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       book = double(:book)
5       order_item = double(:order_item)
6
7       allow(order_item).to receive(:book).and_return(book)
8       → expect(order_item).to receive(:update).and_return(true)
9
10      allow(ImageDownloader).to receive(:call).and_return({ page: 1, images: [] })
11      allow(ImageComposer).to receive(:call).and_return([])
12      allow(PdfGenerator).to receive(:call).and_return(File.open("test_book.pdf", :r))
13
14      BookGenerator.new(order_item).call
15    end
16  end
17 end
```

So that's exactly what we are going to do.

We tell RSpec that we EXPECT the order_item object to receive a call to it's update method.

This is called mocking. You can think of mocks as stubs that HAVE TO BE executed, else the test fails.

MOCKS | STUBS

	Query	Command
Incoming	assert return value	assert side effects
Outgoing	do not assert	assert message sent

Now the question arises of when we should stub and when we should mock behaviour.

This is a table I shnaged from Katrin Owen who shnaged it from Sandy Metz.

First off we can say that in only two kind of method calls exist when considering object interaction. Queries and commands. Queries just return a value. You can think of them as getters or simple questions to an object. While commands change the state of an object. You can think of them as setters, though setters are the most basic type of commands.

We can also differentiate Incoming and outgoing method calls. Incoming are the ones that the object under test receives. While outgoing are ones that are called from the object under test on another object.

MOCKS | STUBS

	Returns value	Changes state
Own method	assert return value	assert side effects
External method	do not assert	assert message sent

This is usually quite confusing and people need some time to wrap their heads around those concepts. So here is a simpler version of the table. You should always assert your own methods and stub or mock method calls to other objects.

MOCKS | STUBS

	Returns value	Changes state
Own method	assert	assert
External method	stub	mock

And here is an even simpler table that exactly specifies when to assert, stub or mock.

MOCKS | STUBS



```
1 class BookGenerator
2   attr_reader :order_item, :book
3
4   def initialize(order_item)
5     @order_item = order_item
6     @book = @order_item.book
7   end
8
9   def call
10    images = ImageDownloader.call(book)
11    composed_images = ImageComposer.call(images)
12    pdf_file = PdfGenerator.call(composed_images)
13    order_item.update(book_pdf: pdf_file)
14  end
15 end
```

The screenshot shows a code editor window titled 'book_generator.rb' with a 'Raw' button. The code defines a `BookGenerator` class. Red annotations are present: three arrows point from the word 'Stub' to the calls to `ImageDownloader.call`, `ImageComposer.call`, and `PdfGenerator.call`. One arrow points from the word 'Mock' to the `order_item.update` call. The code is as follows:

Here's an example.

The calls to `ImageDownloader`, `ImageComposer` and `PdfGenerator` are all calls to external objects that only return a value, therefore we stub them. The update call on `order_item` changes state of the object, therefore we mock it.

MOCKS | STUBS

Isolated testing
Separation of concerns
Speed

Using mocks and stubs you can test your modules in true isolation thus greatly improving you unit testing capabilities. As mentioned before your tests now only test code of the module under test. And a nice side effect of these two is speed. And when I say speed i mean 500 tests in 2 sec.



Spies are an interesting solution to a small problem.

SPIES

```
book_generator_spec.rb

1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       book = double(:book)
5       order_item = double(:order_item)
6
7       allow(order_item).to receive(:book).and_return(book)
8       → expect(order_item).to receive(:update).and_return(true)
9
10      allow(ImageDownloader).to receive(:call).and_return({ page: 1, images: [] })
11      allow(ImageComposer).to receive(:call).and_return([])
12      allow(PdfGenerator).to receive(:call).and_return(File.open("test_book.pdf", :r))
13
14      BookGenerator.new(order_item).call
15    end
16  end
17 end
```

Perhaps you find this syntax unwieldy because there is no explicit `expect` at the end of this spec. Basically your assertion is in the middle of the test. This can be quite confusing when you read this test for the first time.

SPIES

```
book_generator_spec.rb
1 describe BookGenerator do
2   describe '#call' do
3     it "generates and attaches a book's PDF file to the given order item" do
4       book = double(:book)
5       order_item = double(:order_item)
6
7       allow(order_item).to receive(:book).and_return(book)
8       → allow(order_item).to receive(:update).and_return(true)
9
10      allow(ImageDownloader).to receive(:call).and_return({ page: 1, images: [] })
11      allow(ImageComposer).to receive(:call).and_return({})
12      allow(PdfGenerator).to receive(:call).and_return(File.open('test_book.pdf', :r))
13
14      BookGenerator.new(order_item).call
15      → expect(order_item).to have_received(:update)
16    end
17  end
18 end
```

This is the spy syntax.

Please note that doubles are not spies! But can be used as spies. I'll come to that in a moment.

So you create a stub as usual but ask it if a method has been called on it after the module under test gets executed.

SPIES

```
spy.rb
1 describe '#have_received' do
2   it 'expects deliver to get called' do
3     invitation = spy('invitation')
4     invitation.deliver
5     expect(invitation).to have_received(:deliver)
6   end
7 end
```

You can create spies using the `spy` method.

The key difference between doubles and spies is that you can call any method on a spy (implemented / stubbed or not) while calling a non-stubbed method on a double would raise an error.

That's why I said that doubles can be used as spies!



BONUS NACHOS!

Fakes!

A word of warning though, fakes aren't a good idea and can cause a lot of problems so use them only when there is no other possible way.

FAKES

```
checkout_controller_spec.rb
1
2 describe CheckoutController do
3   describe "#checkout" do
4     it 'registers the transaction' do
5       allow(PaymentProcessor).to receive(:register_payment).and_return({ ... })
6     end
7     ...
8   end
9 end
10 end
```

This behaviour can't really be stubbed...

The rest of the code acts differently depending on what payment method was used, if the user was registered, if the user used his account or a payment gateway to pay...

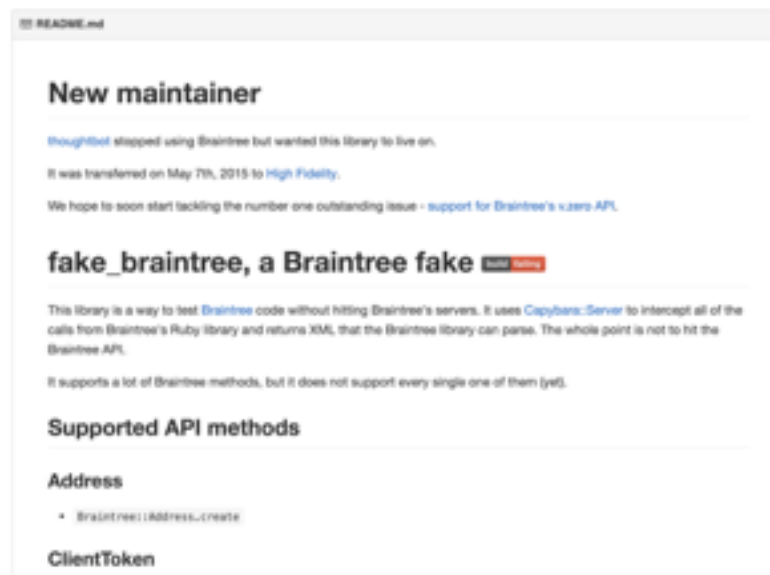
Too many variables, too many states, too many responses.

FAKES

```
85 def self.verify_all_cards!  
86   self.verify_all_cards = true  
87 end  
88  
89 def self.generate_transaction(options = {})  
90   history_item = {  
91     'timestamp' => Time.now,  
92     'amount' => options[:amount],  
93     'status' => options[:status]  
94   }  
95   created_at = options[:created_at] || Time.now  
96   {  
97     'status_history' => [history_item],  
98     'subscription_id' => options[:subscription_id],  
99     'created_at' => created_at,  
100    'amount' => options[:amount]  
101   }  
102 end
```

At this moment it is easier to implement a class that simulates the behaviour of the module or external service.

FAKES



Perhaps one of the most famous fakes is `fake_braintree` which simulates ~80% of braintree's functionality. Offers data persistence and many other features. For those that don't know, braintree is PayPal's API.



Introducing these tool also causes some problems.

"With great power comes great responsibility."

–Ben Parker

Please keep this quote in mind when working with stubs!

```
articles_controller_spec.rb

1 describe ArticlesController do
2   describe '#create' do
3     it 'creates a new object' do
4       article_params = { ... }
5
6       allow(Article).to receive(:create).and_return(true)
7
8       expect(Article.create(article_params)).to eq(true)
9     end
10  end
11 end
```

This is a snippet that i got to review.

If you don't see what's wrong here. Basically we stub the create method on article and assert the same value we set it to return in the following line.


```
articles_controller_spec.rb
1 describe Dashboard do
2   describe "#posts" do
3     it "returns posts visible to the current user" do
4       user = double('user')
5       other_user = double('other user')
6       published_one = double 'post', user: other_user, published: true, title: "published_one"
7       published_two = double 'post', user: other_user, published: true, title: "published_two"
8       unpublished = double 'post', user: other_user, published: false, title: "unpublished"
9       visible_one = double 'post', user: user, published: false, title: "visible_one"
10      visible_two = double 'post', user: user, published: false, title: "visible_two"
11
12      posts = [published_one, published_two, unpublished, visible_one, visible_two]
13
14      allow(Post).to receive(:all).and_return(posts)
15      dashboard = Dashboard.new(posts: Post.all, user: user)
16
17      allow(posts).to receive(:visible_to).with(user).and_return(posts - [unpublished])
18      result = dashboard.posts
19
20      expect(result.map(&:title)).to match_array(%w[
21        published_one
22        published_two
23        visible_one
24        visible_two
25      ])
26    end
27  end
28 end
```

This one is overkill!

The idea is to mock a model.

It doesn't matter what it returns as long as it returns different results for different inputs.

```
articles_controller_spec.rb

1
2 describe Dashboard do
3   describe "#posts" do
4     it "returns posts visible to the current user" do
5       user = double('user')
6       post = double('visible post')
7       posts = [post]
8
9       dashboard = Dashboard.new(posts: posts, user: user)
10
11       allow(posts).to receive(:visible_to).with(user).and_return(posts)
12       expect(dashboard.posts).to match_array(posts)
13     end
14   end
15 end
```

The previous one should be refactored to something like this.

Or even better you could use factory girl to create a stubbed version of the model using `build_stubbed`



Stubs, mocks and spies mostly apply only to unit tests.

Unit tests aren't everything, two completely fine units can fail miserably when combined together.

The picture above is a prime example.

Both windows work fine, but together they are unusable.

Just to be clear I'm talking about physical windows, not the OS (Microsoft Windows).

Well this is a hard one to argue, you simply need integration tests.

They don't need to be complex but they should push a dataset through just to be on the safe side.

It's also arguable that you should know your module's interface.

Well code changes the more people work on it, people forget to update all modules to reflect those changes.

It's better to be safe than sorry.

Yes you will put in extra time to write one more integration test, but you were able to cover much more ground using stubs, mocks and spies than with integration testing.

THANKS FOR LISTENING