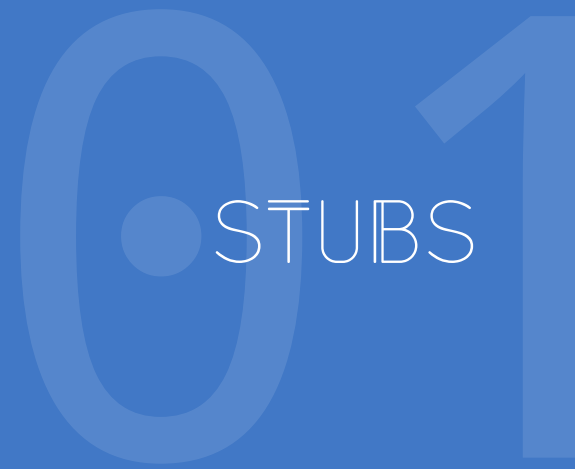


# STUBS | MOCKS | SPIES

I will try to explain to you the difference between stubs, mocks, spies and fakes. Why they are useful and how to use them.



STUBS

# STUBS

book\_generator.rb

Raw

```
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 we have 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

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       BookGenerator.new(order_item).call
6
7       expect(order_item.book_pdf).not_to be_nil
8     end
9   end
10 end
```

Ok! This isn't the best spec in the world but it does the job.

Does anybody see a problem with this spec?

What do you think will happen if we run this spec now?

# STUBS

```
book_generator.rb Raw
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
```

Well it would fail because we haven't yet implemented ImageDownloader, ImageComposer and PdfGenerator! Maybe some of you feel tricked now, imagine then that for any reason those classes aren't working.

# 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       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 spec we actually tested the behaviour of ImageDownloader, ImageComposer, PdfGenerator and finally BookGenerator. This clearly isn't good.

# 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
```

Here is the sam spec using stubs.

We are stubbing the `call` method on each class and telling it what to return when it gets called.

Now our spec will pass as no code outside BookGenerator gets executed.

# STUBS

```
book_generator.rb Raw
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

You can think of stubs as bypass surgery for the module under test.

Instead of going through a submodule's execution path you simply tell it to return something then and there when the desired method gets called.

No code of the sub-module gets executed!



# STUBS

- In computer science, test stubs are programs that simulate the behaviours of software components (or modules) that a module undergoing tests depends on
- They respond only with predefined answers!
- The idea is to make your module's spec independent of other modules thus enabling you to test only the logic in the module under test
- You can write specs that pass if the modules you depend on don't exist or are failing

```
articles_controller_spec.rb
1 describe ArticlesController do
2   describe '#create' do
3     context 'passed valid params' do
4       before do
5         → allow(Article).to receive(:save).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(:save).and_return(false)
16      end
17
18      it 'renders the new page again' do
19        ...
20      end
21    end
22  end
23 end
```

Using stubs you can easily control the module's execution path.

This is most noticeable in controllers when testing a 'happy' and a 'sad' path.

There is no need to craft a good and bad dataset, simply tell the save method to either return true or false

# STUBS

```
articles_controller_spec.rb
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 notice that stubs have a very rich method set with which you can craft exactly the behaviour you want to. In this example the `create` method on the `Article` class will return a different result depending on what arguments get passed to it.

# STUBS

- Independent specs
- Faster specs
- Granular control of the execution path
- Easier collaboration

Using stubs makes your spec independent and faster as they effectively shorten the execution path.

You can easily craft a desired execution path without worrying about and maintaining multiple datasets.

And finally they make collaboration much easier as you can be sure that the logic you wrote works, while other implement modules your code depends on.

# 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
```

But let's get back to this spec for a moment.

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 could turn become a bottleneck in larger test suites.

Perhaps we can fix this?

# 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       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
```

Here we create two empty dummy objects and tell the `order\_item` object what it should do when it receives certain method calls. All of a sudden we stopped depending on the implementation of OrderItem and Book!

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

Oh! Something that confused me terribly when I had my first contact with doubles is the thing in the brackets. What is that? Well that's the name of that particular object! It's used only for debugging messages.

And another thing!

Have you ever wondered how you could test what happens when the destroy method in your controller fails?

Well with stubs you can :)

# DOUBLES

- Empty objects
- Their methods can only be stubbed
- Used in place of other objects (stunt doubles)

Another term closely linked to stubs.

They are empty objects with no methods except for the basic object methods.

They are used instead of other objects for various reasons.

The most common reason is because it's easier to create a double than to instantiate the desired object. In my opinion that's the wrong reason as the primary function of doubles is to replace instances of other classes that may or may not be implemented or faulty thus causing your test to fail...



This brings us to mocks.



# MOCKS

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       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 can't set it using the `update` method as it is stubbed.

# MOCKS

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       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
```

The only sensible way to test if everything went well is to check whether or not the update method got called!

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

And that are mocks in a nutshell!

Mocks are stubs that HAVE TO BE executed, else the spec fails implicitly.

# MOCKS

- They are very similar to stubs
- The key difference is that the spec will check if a mock has been called
- Stubs that have to be executed
- Shares the stub syntax



Spies are an interesting solution to a small problem.

# SPIES

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       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 test is in the middle of the spec, which isn't that logical when reading a spec for the first time.

# SPIES

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       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**

Raw

```
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!

# SPIES

- More explicit version of mocks
- Have a different syntax





#### BONUS NACHOS!

I will now talk about 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

Raw

```
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
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

 README.md

## New maintainer

[thoughtbot](#) stopped using Braintree but wanted this library to live on.

It was transferred on May 7th, 2015 to [High Fidelity](#).

We hope to soon start tackling the number one outstanding issue - [support for Braintree's v.zero API](#).

## fake\_braintree, a Braintree fake

build failing

This library is a way to test [Braintree](#) code without hitting Braintree's servers. It uses [Capybara::Server](#) to intercept all of the calls from Braintree's Ruby library and returns XML that the Braintree library can parse. The whole point is not to hit the Braintree API.

It supports a lot of Braintree methods, but it does not support every single one of them (yet).

## Supported API methods

### Address

- `Braintree::Address.create`

### ClientToken

Perhaps one of the most famous fakes it `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.

# FAKES

- Additional code
- You need specs for your specs
- Unexpected behaviour
- Use webmock or VCR instead

Fakes have many problems.

They are additional code that you need to maintain.

You should write specs for them, but then you are basically writing specs for your specs.

You can never really know how an external service reacts to a given dataset, therefore your implementation can have unexpected behaviour.

It would be better to use the Webmock or VCR gems instead.



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.rbRaw

```
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
```

I think we all see the problem with this one...

It will always pass!

This is the simplest one i have seen, but people get tangled up in such mistakes often when they start using stubs for the first time.



```
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.rbRaw

```
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.



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 as complex as writing a unit-integration test hybrid 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, at least in my humble opinion.

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

[stanko.krtalic@gmail.com](mailto:stanko.krtalic@gmail.com)

@monorkin

<https://github.com/Stankec>