Writing RSpec tests

[Step 1] Describe

While writing any test the first thing you write is the describe block. Let’s suppose we are writing test for a method called **play** inside a **JukeBox** class. First describe block will contain the class name. Describe block helps you know what class/method you are writing test for.

```
describe JukeBox do

end
```

The next thing you write in describe is the method name

```
describe JukeBox do

 describe ‘play’ do

 end

end
```

Note: If your class has just a single method and no more methods will be added in that class you can skip the method describe part.

[Step 2] Context

Once the description about the method for which you are writing the test is there the next step would be to define the context. A method could work differently for different context.

Let’s add two context for the method.

1. When user has inserted coin
2. When user has not inserted coin

```
describe JukeBox do
```

```
  describe ‘play’ do

   context 'when user has inserted coin' do
    end

   context 'when user has not inserted coin' do
   end

 end

end
```

While writing context think about all the possible cases that could come in.

[Step 3] Scenario

Now inside each context write the scenario what it should be doing. In our jukebox when a coin is inserted the jukebox’s play method should return ‘a beautiful song for you’ and when the user has not inserted a coin the jukebox should return ‘no money no song :]’. So let’s add them.

```
describe JukeBox do

  describe ‘play’ do

    context 'when user has inserted coin' do

      it 'return song being played message' do

      end

    end


    context 'when user has not inserted coin' do

      it 'return insert coin message' do

      end

    end

  end

end
```

[Step 4] Subject

Subject is the main part of the test. Write down how you make the method call in a subject. Put the subject at the top so that you can reuse it across different context.

```
describe JukeBox do

  describe ‘play’ do
    subject(:play_music) { described_class.new(coin).play }

    context 'when user has inserted coin' do
      let(:coin) { Coin.new(10) }

      it 'return song being played message' do

      end

    end


    context 'when user has not inserted coin' do
      let(:coin) { FakeCoin.new(10) }

      it 'return insert coin message' do

      end

    end

  end
end
```

Notice that i have added coin variable inside each context since the only thing that changes in these two context is the coin. If you do not define the coin variable it will throw an error that the variable coin is not defined since it is in the used in subject of the test.

[Step 5] Mock/Stub

Now mock/stub any external calls that you don’t need to cover test for. There might be some other methods being called, or any external api calls or even database calls. Let’s say this play method uses validate_coin method to know if the coin is valid or not.

```
describe JukeBox do

  describe ‘.play’ do
```

```
    subject(:play_music) { described_class.new(coin).play }

    context 'when user has inserted coin' do
      let(:coin) { Coin.new(10) }

      before do
        allow(JukeBox).to_receive(:validate_coin)
          .with(coin).and_return(true)
      end

      it 'return song being played message' do

      end

    end


    context 'when user has not inserted coin' do
      let(:coin) { FakeCoin.new(10) }

      before do
        allow(JukeBox).to_receive(:validate_coin)
          .with(coin).and_return(false)
      end

      it 'return insert coin message' do

      end

    end

  end

end
```

[Step 6] Expectation

Now everything is set. The only thing remaining is to write the actual expectation.

```
describe JukeBox do

  describe ‘play’ do
    subject(:play_music) { described_class.new(coin).play }

    context 'when user has inserted coin' do
      let(:coin) { Coin.new(10) }

      before do
        allow(JukeBox).to_receive(:validate_coin)
          .with(coin).and_return(true)
      end

      it 'return song being played message' do
```

```
        expect(play_music).to return('a beautiful song for you')
      end

    end


    context 'when user has not inserted coin' do
      let(:coin) { FakeCoin.new(10) }

      before do
        allow(JukeBox).to_receive(:validate_coin)
          .with(coin).and_return(false)
      end

      it 'return insert coin message' do
        expect(play_music).to return('no money no song :]')
      end

    end

  end

end
```

All set. The test is now ready.

[Step 7] Refactor

The test is now running well. There seems to be some code that would be reused. So let’s use a variable and reuse the before each block.

```
describe JukeBox do

  describe ‘.play’ do
    subject(:play_music) { described_class.new(coin).play }

    before do
      allow(JukeBox).to_receive(:validate_coin)
        .with(coin).and_return(valid)
    end

    context 'when user has inserted coin' do
      let(:coin) { Coin.new(10) }
      let(:valid) { true }

      it 'return song being played message' do
        expect(play_music).to return('a beautiful song for you')
      end

    end
    context 'when user has not inserted coin' do
      let(:coin) { FakeCoin.new(10) }
      let(:valid) { false }

      it 'return insert coin message' do
        expect(play_music).to return('no money no song :]')
      end

    end

  end

end
```

With this there will be a fixed pattern of writing test. If all developers across the organisation follow the same structure it will be easy for all the developer to modify the code that someone else has written. Also if a developer wants to use a method that is already there, test helps to give the context what is the expected response of that method and what are the parameters that it takes.