Testing

Writing automated tests for the code is essential. Ruby community emphasizes it and most of projects are well covered. A TDD is also popular among rubyist.

Testing frameworks

Today de facto standard is to use Minitest testing framework. You can see RSpec being still used too but Minitest already offers the capabilities and more. Both can be easily used for TDD. For BDD there's popular ecosystem called Cucumber which started as Ruby gem but quickly evolved into polyglot tool.

Minitest

A syntax you can see in tests can be in two forms. Either something we call testunit (aka junit) and spec that was taken from Rspec. The internal implementaion is the same for both and it's mostly matter of taste. However tests are regular ruby scripts that test other scripts. Test are usually to be found at `test/` directory, file name should reflect test class defined inside, e.g. morse_coder_test.rb.

Example output of

Testunit syntax

```
require 'minitest/autorun'
require 'morse_coder.rb'

class MorseCoderTest < Minitest::Test
  def setup
    @coder = MorseCoder.new(...)
  end

  def test_encode_for_single_letters
    assert_equal ".-", @coder.encode "a"
    assert_equal "-...", @coder.encode "b"
  end
end
```

Test class should inherit from Minitest::Test so test helpers (assertions) are available. Testing methods must start with "test_". Other methods are regular methods that can be used from testing methods, e.g. to build some fixtures.

There are few method names with special meaning. In the example you can see method with name `setup`. This method gets automatically executed before every testing method. Similarly there's `teardown` method that get's executed after each testing method. It's usually used for cleaning up mess the testing method created.

Assertions

One testing method can contain more than one assertion. First assertion failure stops the method run and mark the test as failure (F). If method raises an exception the result of test is marked as error (E). If all assertions defined in method passes, test succeeds (.). If you plan to implement the test later you can skip the test by calling `skip("Sorry, I’m lazy")`.

The simplest assertion is to test boolean value like this

```
assert @something
```

This will succeed if @something is considered true, fail otherwise. The negative form is refute, e.g. following would pass.

```
refute false
```

You could obvisouly add tests like `assert @something == 'that I expect'` but it would generate very generic messages on failures. You can specify custome message by passing extra argument like this

```
assert @something == 'that I expect', '@something does not match expected string'
```

but it’s always better to use assert helper that matches the use-case best. Following example demonstrates how to check equality of two values, the failure message would automatically include information about what’s it @something and what was expected it to be.

```
assert_equal @something, 'that I expect'
```

Useful assert helpers are listed in example below. All of them can be found in Minitest documentation.

```
assert arg             # arg is true
refute arg             # arg is false

assert_equal           expected, actual
assert_includes        collection, object
assert_kind_of         class, object
assert_nil             object
assert_match           expected, actual
assert_raises          exception_class &block
```

Spec syntax

Subjectively better structured, less repeating, more readable and TDD supporting syntax can be used. See the following example.

```
require 'minitest/autorun'
require 'morse_coder.rb'

describe MorseCoder do
  let(:coder) { MorseCoder.new(...) }

  describe 'single letters encoding' do
    let(:a) { coder.a }
    let(:b) { coder.b }

    specify { a.must_equal '.-' }
    specify { b.must_equal '-...' }
  end
end
```

Describe block wraps logical block. Each such block can have it's own `before` (aka setup). With `let` we define method that can be called later within any nested block. Let is lazy. `specify` accepts a lock that uses assertion helpers in form of `must_$assert` or `wont_$assert`. There are many other extensions to this language so it reads more naturally.

Note that since the implementation is the same, you can combine both at the same time.

Output of test run

```
Run options: --seed 25127

# Running tests:

..S.....F........

Finished tests in 101.524752s, 6.4319 tests/s, 9.0618 assertions/s
63 tests, 92 assertions, 1 failures, 0 errors, S skips

  1) Failure:
TestConnector#test_connection [./connector.rb:5]:
  Expected: nil
  Actual: "that I expect"
```

The seed is random number representing the order of test. Note that your tests should be order indpenendant.

Running multiple test files

It's common to have more than just one test file in project. To run all tests at once we can use Rake. Usually tests are put in `test` directory in the project tree structure. In such setup we can easily define test task in Rakefile. Rake provides built-in class for this, we just need to configure it. Just put following into your Rakefile.

```
require 'rake/testtask'

Rake::TestTask.new do |t|
  t.libs << 'test'
  t.test_files = Dir.glob('test/**/*_test.rb')
  t.verbose true
end
```

we can run `rake test` which will load a run all ruby scripts with `_test` suffix in the test directory including all of its subdirectories. If you prefer test to be the default rake task, add following to the Rakefile

```
task :default => [ :test ]
```

now you can run all tests just by running `rake`.

Another common practise is to have one file that is loaded at start, usually named test_helper.rb. This file contains everything that is needed for all tests, like requiring additional testing libraries. You can also put `require minitest/autorun` there. Just note that you need to `require 'test_helper'` as first line of every test file.

Test coverage

To get a good overview of what needs test coverage it's useful to setup code coverage check. A simplecov gem can generate html report. Just put following on top of you test_helper.rb

```
require 'simplecov'
SimpleCov.start
```

you can also define a minimum coverage in percents

```
SimpleCov.minimum_coverage 95
```

Now when you run your test suite, new directory called `coverage` will be created. See `coverage/index.html` for details how well your code is covered with tests.

Stubbing

Sometimes we don't want to call all method chain when we test just single method behavior. This applies especially in unit testing where we test just small piece of code. Since Ruby is dynamic language, it's easy to cut off some methods. This is called stubbing (leaving stubs).

Let's look at following example

```
class TemperatureMeter
  def measure(output)
    temp = rand(21) + 20
    output.puts temp
    temp
  end
end
```

The test covering this should call method measure and verify it returns reasonable temperature. We don't want our test to print anything to STDOUT. We can stub out puts method easily like this

```
def test_measure
  meter = TemperatureMeter.new
  STDOUT.stub(:puts, nil) do
    result = meter.measure(STDOUT)
    assert_kind_of Fixnum, result
    assert_includes 20..40, result
  end
end
```

With this stubbing, `puts` method is replaced by new empty method that returns the second argument, in this case `nil`. The stub is applied only within the stub block.

Mocking

Mocking is related to stubbing. Imagine we wanted to check that measure method really called puts on output object. The method is written in a way that it accepts custom output object, which makes testing easy. We can simply pass any object that implements method `puts`, e.g. file handler, socket or our own testing object. Or we can use mocks. Mock is a blank object on which we can define expectations.

For example we can create a mock instance and specify that its method puts should be called exactly once during the test.

```
def test_measure_print_the_value
  meter = TemperatureMeter.new
  mock = Minitest::Mock.new
  mock.expect(:puts, nil, [20..40])
  result = meter.measure(mock)
  mock.verify
end
```

First `expect` argument is the name of method to be called, second is the return value and third is the array containing arguments which the puts should be called.

You could also stub the `rand` method to return let's say `0` and then setup expectation that mock's `puts` method will receive `20` as a parameter to print. But the range also works so the mock accepts any value between `20` and `40`.

You have to call verify on mock so it runs assertions on how many times the expected method was called. To expect another call of puts, just define new expectation with `.expect`.

Stubbing network calls

If your app communicates with external services over HTTP you most likely need to fake the communication in your test suite. Reasons include performance, spamming of remote services, avoiding credentials leaks, error state testing. Constructing the whole net/http response object can be complicated. Luckily there are tools that can help you greatly.

First is webmock gem. It provides helpers to stub low-level methods easily. To use it, install the gem and just add following to your tests.

```
require 'webmock/minitest'
stub_request(:get, 'www.example.com')

Net::HTTP.get('www.example.com', '/') # this will succeed
```

You can also specify more conditions to match the request as well as return value

```
stub_request(:post, 'www.example.com').with(:body => 'ping').to_return(:body =>
'pong')
```

Custome headers can be added too. Webmock works with higl level libraries such as popular Restclient gem.

Another useful tool is vcr. The name was chosen because of analogy with videocassette recorder. It can record a real network communication and replay it later. This can be nicely used in tests. You only record the communication once during writing tests and replay it while running tests in future or on CI server. You can have multiple communications recorded and just swap cassetes for each test. Example follows

```
require 'vcr'

VCR.configure do |config|
  config.cassette_library_dir = "fixtures/vcr_cassettes"   # storage for cassetes
  config.hook_into :webmock                                 # webmock integration
end

class VCRTest < Minitest::Test
  def test_example_dot_com
    VCR.use_cassette("success_info") do
      response = Net::HTTP.get_response(URI('http://www.example.com/'))
      assert_match /Example Domain/, response.body
    end
  end
end
```

Testing web applications

If you work on web app you can also easily test the interaction like users will interact through web browser. This is useful when you write integration tests. A de facto standard is capybara gem that provides drivers for various browser backends. The simplest to setup driver is RackTest, so you can start with it as long as your app uses rack.

If you need advanced stuff like testing pages with asynchronous requests through AJAX you can use Selenium driver which runs firefox in headless mode. If you want to run such tests on CI server without X11 server, there’s Poltergeist driver using PhantomJS.

An example of simple test, supposing my_app.rb contains rack based app (e.g. using Sinatra).

```
require 'minitest/autorun'
require 'capybara/dsl'
require './my_app.rb'

Capybara.app = MyApp
Capybara.default_driver = :rack_test

class MyAppTest < Minitest::Test
  include Capybara::DSL

  def test_index
    visit '/'
    click_link 'login'
    fill_in('Login', with: 'Marek')
    fill_in('Password', with: 'secret')
    click_button('Submit')

    assert page.has_selector('div p.success')
    assert page.has_content?('Welcome Marek')
  end

  def teardown
    Capybara.reset_sessions!
    Capybara.use_default_driver
  end
end
```

Cucumber

We can use Cucumber framework for BDD aproach. It allows us to write the behavior specification in natural language first and then convert it to test step by step. Imagine you'd describe a feature like this

```
Feature: logout of logged in user

  Scenario: User can log out from app
    Given I'm logged in as user ares
      And I'm on host list page
    When I click logout link
    Then I should see logout notification
```

It's a valid cucumber test (aka feature) which only needs implementing those steps, usign capybara for example.

```
Given(/^I'm logged in as user (.*)$/) do |user|
  visit '/'
  fill_in "login", with: user
  fill_in "password", with: 'testpassword'
  click_button 'login'
end

Given(/^I'm on (.*) (.*) page$/) do |resource, action|
  visit "/#{resource}/#{action}"
end

When(/^I click (.*) link$/) do |identifier|
  click_link identifier
end

Then(/^I should see logout notification$/) do
  assert page.has_content 'div p.logout_notification'
end
```

One advantage that it brigns is, that your tests are live documentation too.