9

Testing with IronRuby

* Learn about Ruby testing tools
* Learn about CLR interop rules
* Learn about Caricature

At the start of this book we mentioned testing as a very important part of the development process, but the book has been rather thin on testing. That is about to change we’re going to have an exploration of testing both Ruby objects and CLR objects in this chapter. When I got first introduced to the concept of unit testing a few years ago I had to overcome a great deal of resistance. As soon as I got the hang of it and developed an application with a test-first approach I was sold on the concept.

As I’ve mentioned before testing gives me a warm fuzzy feeling and allows me to refactor in confidence. It’s also a good communication tool for collaborating with other members in your team as they immediately have a documented form of how the code is supposed to work. There is an entire book dedicated to testing with RSpec and Cucumber and the subject of BDD on itself is large enough to write one or several books on. We will just look at using these tools in the context of pure (Iron)Ruby and IronRuby to CLR interoperability.

At first sight testing stuff with ruby should be pretty easy because of its dynamic nature you have a great deal of control on how your objects look when you start testing them. However when you start dealing with objects defined with a CLR language the CLR interop rules kick in which may be confusing when you’re approaching problems with a dynamic mindset. At the very least you can get some surprising situations if you don’t follow those rules.

It took me a while to decide what I was going to put in this chapter because providing yet another tutorial on how you use RSpec and Cucumber didn’t feel right to me. I’m also not talking about cucumber because at this point I don’t consider it something you can rely on heavily when using Ironruby. Instead I’ve opted to take the mocking library I wrote as an example so that we can look at the items that make IronRuby a special case. Before we start that it might be helpful to give an overview of the most popular tools from the ruby community for testing.

9.1. A map to the ruby testing jungle

The ruby community has been on the forefront of the evolution of TDD and BDD. Lots of people chose ruby as a language to experiment with and toss ideas around because of its flexibility and malleability. Because of this, testing is one part of the ecosystem that has been more than adequately catered for. This section aims to give you a quick overview of the tools available to you and you’re free to pick whatever tool appeals most to you. If you want to follow along in this chapter you’ll need to install a few gems. Executing igem install cucumber shoulda bacon rspec mocha caricature will install the necessary gems. We’ll start with tools for unit testing, moving on to acceptance test tools and finally mocking libraries.

9.1.1 Low hanging fruit: unit tests

A great indication of how much importance the ruby community puts into testing is the fact that there is a unit test framework included in the standard libraries for Ruby. This framework is called Test::Unit and follows the xUnit standard like jUnit, NUnit, … Table 9.1 shows an overview of some of the tools available with a short description. This is by no means an exhaustive list of frameworks last time I checked there were at 20 test frameworks for the Ruby language.

|  |  |  |
| --- | --- | --- |
| Name | Description | URL |
| Test::Unit | xUnit standard. Included in the ruby standard lib | http://ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html |
| Shoulda | Brings context/specification style testing to Test::Unit. Great rails integration | http://github.com/thoughtbot/shoulda |
| Bacon | Context/specification framework. Very lightweight Very lightweight | http://github.com/chneukirchen/bacon |
| RSpec | The original BDD context / specification test framework for Ruby | http://rspec.info/ |

Table 9.1: A few test frameworks

Most of the other libraries don’t really bring anything new to the table but rather focus on solving the same problem but with a different name for methods. From a TDD perspective you’re probably best of sticking to plain old Test::Unit. If you have experience with NUnit you should be able to pick that up in no time. Listing 9.1 shows a test for an add method written with Test::Unit.

Listing 9.1: Example of a Test::Unit test

require 'rubygems' unless defined?(Gem)

**require 'test/unit'**

require File.dirname(\_\_FILE\_\_) + '/calculator.rb'

class TestAddition < Test::Unit::TestCase

def **setup**

@calc = CalculatorOperation.new(2)

end

def **test\_**add

res = @calc.add(2)

assert\_equal 4, res

end

def **test\_**substract

res = @calc.sub(1)

assert\_equal 1, res

end

end

However if you want to do BDD you can stick to Test::Unit and use Shoulda to write your tests in a context/specification style. Listing 9.2 shows you the test for the add method but with Shoulda. One of the great advantages of Shoulda is that it has great integration for the Rails framework.

Listing 9.2: Example of a Should test

require 'rubygems' unless defined?(Gem)

**require 'shoulda'**

require File.dirname(\_\_FILE\_\_) + '/calculator.rb'

class TestAddition < Test::Unit::TestCase

**context** "A calculator operation" do

**setup** do

@calc = CalculatorOperation.new(2)

end

**should** "add a second number correctly" do

res = @calc.add 2

**assert**\_equal 4, res

end

**should** "substract a second number correctly" do

res = @calc.sub 1

**assert**\_equal 1, res

end

end

end

Another option for doing BDD is Bacon. I’m mentioning Bacon because the Merb team recently started using bacon and the tests you write with bacon are very similar to the ones you’d write with RSpec. Bacon is useful in the context of IronRuby because the execution of the tests is a lot faster than when you would use RSpec at this point. This may change in the future though as IronRuby gets more performance optimizations. Listing 9.3 shows you the add method test but written with Bacon.

Listing 9.3: A bacon test

require 'rubygems' unless defined?(Gem)

**require 'bacon'**

require File.dirname(\_\_FILE\_\_) + '/calculator.rb'

**describe** "A calculator operation" do

**before** do

@calc = CalculatorOperation.new(4)

end

**it** "should add 2 numbers correctly" do

@calc.add(4).**should** == 8

end

**it** "should subtract 2 numbers correctly" do

@calc.sub(2).**should** == 2

end

end

RSpec is the original BDD context/specification framework, it certainly is my favorite framework for writing specs in Ruby. RSpec is a bit more heavyweight than Bacon but it also has more features, more wide spread adoption and it has a book that documents the testing cycle really well (<http://www.pragprog.com/titles/achbd/the-rspec-book>). RSpec is properly documented and there are many examples available online. Both bacon and shoulda have at least one shortcoming if you ask me. They only allow you to specify a setup for each test method and not one that executes once for a given context.

Listing 9.4: A RSpec example

require 'rubygems' unless defined?(Gem)

**require 'spec'**

require File.dirname(\_\_FILE\_\_) + '/calculator.rb'

**describe** "A calculator operation" do

**before(:all)** do

@calc = CalculatorOperation.new(4)

end

**it** "should add 2 numbers correctly" do

@calc.add(4).**should** == 8

end

**it** "should subtract 2 numbers correctly" do

@calc.sub(2).**should** == 2

end

end

Personally I prefer the BDD style of writing tests, this is a bit subjective and your mileage may vary. But when I was writing unit tests I felt a kind of pressure to test every little thing in my application. Write tests for private methods etc. With BDD it starts with: you test the behavior of an object hence the pressure of testing everything moves to testing the public interface of a class. We’ve just seen a couple of tools we can use to do class level tests. The next section will talk about application level tests.

9.1.2 Higher hanging fruit: acceptance tests

Being able to do class level verifications is only one piece of the puzzle we also want to ensure that we’re working on the right stuff. One tool to make sure you’re doing that is to use acceptance tests in which you write down a bunch of stories that you can then use to test your application as a whole. Ruby is again on the frontline here and has a few great libraries to help you do that. By far the most popular project in this space is cucumber.

Cucumber uses an external DSL called Gherkin to define the user stories. These stories get then converted to runnable step definitions. These step definitions execute the user stories defined in the feature files. Listing 9.5 shows an example of a feature written in Gherkin.

Listing 9.5: The acceptance test for the calculator

Feature: Addition

In order to avoid silly mistakes

As a calculator user

I want to be told the sum of two numbers

Scenario: Add two numbers

**Given** I visit the calculator page

And I fill in '**50**' for '***left***'

And I fill in '**70**' for '***right***'

**When** I press '**Add**'

**Then** I should see '**Answer: 120**'

Of course this feature file has steps to go with it, these steps are shown in listing 9.6.

Listing 9.6 The step definitions for the cucumber feature

**Given** /^I visit the calculator page$/ do

visit '/add'

end

**Given** /^I fill in '**(.\*)**' for '***(.\*)***'$/ do |**value**, ***field***|

fill\_in(field, :with => value)

end

**When** /^I press '**(.\*)**'$/ do |**name**|

click\_button(name)

end

**Then** /^I should see '**(.\*)**'$/ do |**text**|

response\_body.should contain(/#{text}/m)

end

In this case we’re testing a Sinatra application. So this set of step definitions will automate the browser and test the application that way. This is a very nice way of automating acceptance tests and sharing a common language between the business and the development team. The last set of tools I’d like to highlight are the mocking frameworks.

9.1.3 Make-believe fruits: mockers

Mocking with Ruby is easy or at least it should be easy. After all classes are open you can do whatever you want with them. You also don’t want every test to become an integration test so you want to isolate behavior and at that point you can hand-roll your doubles or use a mocking framework.

Because creating a mocker is easy to do with Ruby there are many of them out there but there are only a few really popular ones. The one that I encounter the most when I get into a ruby project started by somebody else is Mocha. Mocha does exactly what you’d expect a mocking library to do, its syntax is succinct and it allows you to do standard mocks or partial mocks. Listing 9.7 shows a short example of using mocha, it has all the methods related to mocking bolded.

Listing 9.8: An example of a mock with mocha

require 'rubygems' unless defined? Gem

**require 'spec'**

**require 'mocha'**

class CookieJar

def swipe\_cookie

# complex logic here that swipes a cookie

end

end

**Spec::Runner.configure do |config|**

**config.mock\_with :mocha**

**end**

describe "Mocking" do

it "should be caught" do

cookie\_jar = CookieJar.new

msg = "Caught with your fingers in the cookie jar"

**cookie\_jar.expects(:swipe\_cookie).returns(msg)**

*cookie\_jar.swipe\_cookie.should == msg*

end

it "should fail" do

cookie\_jar = CookieJar.new

msg = "Caught with your fingers in the cookie jar"

**cookie\_jar.expects(:swipe\_cookie).returns(msg)**

# Mocha::ExpectationError in 'Mocking should fail'

# not all expectations were satisfied

# unsatisfied expectations:

# - expected exactly once, not yet invoked:   
 # #<CookieJar:0x100cbdd08>.swipe\_cookie(any\_parameters)

end

end

RSpec comes with its own mocking framework but allows you to use your favorite framework. From an API point of view RSpec mocks are very similar to mocha objects. The method names differ and the way they do their job internally is different too. Listing 9.8 shows a short example using RSpec syntax.

Listing 9.8: An example of mocking with RSpec

require 'rubygems' unless defined? Gem

**require 'spec'**

class CookieJar

def swipe\_cookie

# complex logic here that swipes a cookie

end

end

describe "Mocking with RSpec" do

it "should be caught" do

cookie\_jar = **mock(CookieJar.new)**

msg = "Caught with your fingers in the cookie jar"

cookie\_jar.**should\_receive(:swipe\_cookie) { msg }**

cookie\_jar.swipe\_cookie.should == msg

end

it "should fail" do

cookie\_jar = mock(CookieJar.new)

msg = "Caught with your fingers in the cookie jar"

cookie\_jar.**should\_receive(:swipe\_cookie) { msg }**

# Spec::Mocks::MockExpectationError in 'Mocking should fail'

# Mock #<CookieJar:0x00000101396f90> expected :swipe\_cookie   
 # with (any args) once, but received it 0 times

end

end

Both of these mockers are really great libraries but when you’re going to deal with CLR classes there be dragons. These frameworks have been written for Ruby and for Ruby-to-Ruby interactions. And as long as you stick to that you should be fine with any of those libraries. However if you want to mock code you wrote in C# to interact with other code written in VB.NET or something then you will need a mocking library that also knows about the CLR rules.

I wrote such a library when I found out this problem existed, and I called the library Caricature. You can find it on github and it’s distributed as a gem. Listing 9.9 has the example written with caricature.

Listing 9.9: An example of mocking with Caricature

require 'rubygems' unless defined? Gem

**require 'spec'**

**require 'caricature'**

class CookieJar

def swipe\_cookie

# complex logic here that swipes a cookie

end

end

**Spec::Runner.configure do |config|**

**config.mock\_with Caricature::RSpecAdapter**

**config.include Caricature::RSpecMatchers**

**end**

describe "Mocking" do

it "should be caught" do

cookie\_jar = **isolate CookieJar**

msg = "Caught with your fingers in the cookie jar"

cookie\_jar.**when\_receiving(:swipe\_cookie).return(msg)**

cookie\_jar.swipe\_cookie.should == msg

cookie\_jar.should **have\_received(:swipe\_cookie)**

end

it "should **pass**" do

cookie\_jar = **isolate** **CookieJar**

msg = "Caught with your fingers in the cookie jar"

cookie\_jar.**when\_receiving(:swipe\_cookie).return(msg)**

end

it "should **fail**" do

cookie\_jar = **isolate** **CookieJar**

msg = "Caught with your fingers in the cookie jar"

cookie\_jar.**when\_receiving(:swipe\_cookie).return(msg)**

cookie\_jar.should **have\_received(:swipe\_cookie)**

# 'Mocking should fail' FAILED

# expected #<CookieJar541...:0x00000100f40f78> to have received swipe\_cookie

# Couldn't find a method call with name swipe\_cookie

end

end

The biggest difference between caricature and the other mocking frameworks is that it blurs the line between terminologies like mock/stub/… and just uses 1 term for them: isolation. The example with Caricature holds one more spec because it has a different behavior than the traditional mockers. It’s my opinion that in most cases you’re not actually interested in the return values or if certain methods have been called. You’re just interested in testing the item and you want to get there with as easily as possible. For me expectations get in the way many times and I just want to verify certain method calls not all of them so those need explicit declaration.

I’ve left out a bunch of other mocking libraries like rr and flexmock. The main point of this section is that you’re fine with the ruby mocking libraries if you’re not going to interact with the CLR. Before we get to actually testing objects there is one more compelling reason to use IronRuby for testing CLR assemblies: autotest/autospec.

9.1.4. Automating test runs

In the ruby gem ZenTest you will find a few things that make testing with Ruby almost a fun exercise. When you’re writing code and tests it helps a lot if you get into some kind of rhythm and manually executing rake spec or something equivalent can get old really quick. The thing is that for most programmers testing is a real chore and nobody really likes to do it. Write test, fail, write code, pass, refactor, repeat,.. If you then have to break out your development environment and go somewhere else to run tests, which may take a while to load can quickly break that flow because having to wait equals blog reading etc. So the ZenTest library has an autotest executable, which will load your specs and every time you change a file in one of the designated folders it will run the specs. If a spec fails then it will be run first next time you save. When that failing spec now passes it will run the entire batch of specs. The important thing is that it keeps running while your developing. I will have autotest on most of the time when I’m developing with IronRuby/Ruby and when it fails it makes a long and loud beep so I know something is up and can fix the tests.

Throughout the remainder of this chapter we will be going through some of the code in the tests for caricature and explain the challenges you’re going to encounter when testing CLR assemblies. The active specs for caricature are the ones that use bacon. But for this book I’ve converted them to RSpec, so for the rest of this chapter we’ll use the RSpec BDD framework to write our specs. This conversion was really easy and mostly consisted of replacing spaces with dots. Let’s get down to business and start testing some ruby objects.

9.2 Appetizers: ruby-to-ruby interactions

As mentioned earlier mocking ruby objects or creating a mocker for ruby objects is relatively easy because Ruby doesn’t have as many rules as it’s brothers on the CLR. Exactly for this reason we’ll start here and work our way up. This gives me the opportunity to show a little bit of RSpec usage as well as explain some of the API of caricature before we get into the CLR interoperability.

It goes against the basic idea of BDD to write your class before you have specs defining the functionality you’re going to implement. If I would follow the red/green/refactor mantra then this chapter would get a lot longer. We will first define some models because the purpose is to show you how you can test with IronRuby but not to teach you how to do BDD.

9.2.1 The models

Because we want to mimic a more “real world”-like scenario we’ll need a few models that take dependencies on each other. Preferably we have some inheritance going on and at some point we probably also want to include a module of some kind. Listing 9.10 shows some of the models you can find in spec/fixtures in the code samples for this chapter.

Listing 9.10: The models

class Soldier

def initialize

@life = 10

end

def name

"Tommy Boy"

end

def to\_s

"I'm a soldier"

end

**def attack(target, weapon)**

**weapon.attack(target)**

**end**

**def is\_killed\_by?(weapon)**

**weapon.damage > 3**

**end**

**def survive\_attack\_with(weapon)**

**@life - weapon.damage**

**end**

end

class SoldierWithClassMembers **< Soldier**

def self.class\_name

"SoldierWithClassMembers"

end

end

class Dagger

def damage

2

end

**def attack(target)**

**target.survive\_attack\_with self**

**end**

end

module PureRubyModule

end

module RubyModuleIncludingModule

include PureRubyModule

end

class ModuleIncludingClass

**include RubyModuleIncludingModule**

end

There is not much magic going on in these classes, but we do have a mix of things that can confuse a mocker. The module stuff actually doesn’t matter much at this point but it will matter when we’re dealing with the CLR later.

If you’re familiar with mocking frameworks for the CLR like Moq or Rhino.Mocks then you’ll know that these frameworks require you to tell the framework which type you’re want to mock. Caricature adheres to the same principle, as we’ll see next.

9.2.2 The simplest cases

The simplest case in the context of caricature is to create an isolation when it has been passed a class with only instance members. The specs we’ll run need to satisfy a number of conditions, they are outlined below:

Ruby to Ruby interactions when isolating Ruby classes

- should work without expectations

- should work for expectations with an argument constraint

- should work for expectations with an argument constraint when a wrong argument is passed in

- should work for expectations with an argument constraint and an assertion argument constraint

- should fail for expectations with an argument constraint and an assertion argument constraint

- should work with an expectation for any arguments

- should work with an assertion for specific arguments

- should fail for an assertion with wrong arguments

- should execute a callback when an expectation is being invoked and with is not defined in a block

- should execute a callback when an expectation is being invoked and with is defined in a block

If the logger lives up to these expectations we can be reasonably sure we can use it for some real work. Now how do the specs look that generated this output? Listing 9.11 holds the answer.

Listing 9.11: The specs for isolating ruby classes

describe "Ruby to Ruby interactions" do

describe "when isolating Ruby classes" do

before do

@dagger = Dagger.new

@soldier = isolate Soldier

end

it "should work without expectations" do

result = @dagger.attack @soldier

result.should be\_nil

@soldier.should have\_received(:survive\_attack\_with, @dagger) 1

end

it "should work for expectations with an argument constraint" do

@soldier.when\_receiving(:survive\_attack\_with).with(@dagger).return(5) 2

@dagger.attack(@soldier).should == 5

@soldier.should have\_received(:survive\_attack\_with) 3

end

it "should work for expectations with an argument constraint when a wrong argument is passed in" do

@soldier.when\_receiving(:survive\_attack\_with).with(@dagger).return(5)

@dagger.attack(Soldier.new).should\_not == 5

@soldier.should\_not have\_received(:survive\_attack\_with, @dagger)

end

it "should work for expectations with an argument constraint and an assertion argument constraint" do

soldier = Soldier.new

@soldier.when\_receiving(:survive\_attack\_with).with(@dagger).return(5)

@dagger.attack(@soldier).should == 5

@soldier.should have\_received(:survive\_attack\_with).with(@dagger)

end

it "should fail for expectations with an argument constraint and an assertion argument constraint" do

soldier = Soldier.new

@soldier.when\_receiving(:survive\_attack\_with).with(@dagger).return(5)

@dagger.attack(@soldier).should == 5

@soldier.should\_not have\_received(:survive\_attack\_with, Dagger.new)

end

it "should work with an expectation for any arguments" do

@soldier.when\_receiving(:survive\_attack\_with).return(5)

result = @dagger.attack @soldier

result.should == 5

@soldier.should have\_received(:survive\_attack\_with, :any)

end

it "should work with an assertion for specific arguments" do

@soldier.when\_receiving(:survive\_attack\_with) do |method\_should| 4

method\_should.return(5)

end

result = @dagger.attack @soldier

result.should == 5

@soldier.should have\_received(:survive\_attack\_with).with(@dagger)

end

it "should fail for an assertion with wrong arguments" do

@soldier.when\_receiving(:survive\_attack\_with) do |method\_should|

method\_should.return(5)

end

result = @dagger.attack @soldier

result.should == 5

@soldier.should\_not have\_received(:survive\_attack\_with, isolate(Dagger))

end

it "should execute a callback when an expectation is being invoked and with is not defined in a block" do

iso = isolate Dagger

cnt = 0

iso.when\_receiving(:damage).with(:any) do |\*args| 5

cnt += 1

end

iso.damage

cnt.should == 1

end

it "should execute a callback when an expectation is being invoked and with is defined in a block" do

cnt = 0

iso = Caricature::Isolation.for(Dagger)

iso.when\_receiving(:damage) do |exp| 5

exp.with(:any) do |\*args|

cnt += 1

end

end

iso.damage

cnt.should == 1

end

end

end

1. Verify an expectation with arguments
2. Set up an expectation with argument constraint and return value
3. Verify an expectation with any/without arguments
4. Setup expectation by using a configuration block
5. Setup a callback to be executed

The listing we’ve just seen contains all the specs needed to get to the output shown at the beginning of this section. In the first spec we don’t set up any expectations and verify if our mock/isolation always returns the default value, which is nil in this case. We then verify if the method call has been recorded (#1), when doing so we also check if the argument was correct.

The next bit of interest is setting up an expectation with an argument constraint and a return value (#2), then we execute some code and we verify if the method has been called at all (#3).

Instead of using the fluent API to configure an isolation, you can also use a block to configure the same parameters (#4). And lastly there are some use cases where you want to inject a callback of some sort into the call chain (#5). We can do this by using the fluent API or with a configuration block. This shows us the basics of testing ruby stuff while using caricature as your mocker, there are more variations possible like a ruby class with class members or isolating a ruby instance of an object, we’ll look at those now.

9.2.3 Raising the bar a little

The example from the current listing to me is equivalent to a noisy version of hello world. While you would be able to do your job you wouldn’t be doing a very efficient one. At this point there is no way we can reuse the specs we wrote in listing 9.11. We’re going to create specs for 2 variations on the shape of the object we’ll isolate and these objects should also pass the baseline examples we created in listing 9.11. RSpec has the ability to use shared examples so you can reuse a set of examples in different contexts. Conceptually it works a lot like a Module that you include in a class.

So how would we go about making the examples we saw earlier into a shared example group? This is quite easy really: you take the examples you want to share and put them in a block for the shared\_examples\_for method. Listing 9.12 shows the listing for the specs for our new variations and the spec from listing 9.11. Most of the common specs have been clipped to preserve space.

Listing 9.12: Sharing examples

require File.dirname(\_\_FILE\_\_) + "/../spec\_helper"

**shared\_examples\_for "a ruby to ruby interaction"** do 1

it "should work without expectations" do

result = @dagger.attack @soldier

result.should be\_nil

@soldier.should have\_received(:survive\_attack\_with, @dagger)

end

it "should work for expectations with an argument constraint" do

@soldier.when\_receiving(:survive\_attack\_with).with(@dagger).return(5)

@dagger.attack(@soldier).should == 5

@soldier.should have\_received(:survive\_attack\_with)

end

# there were more examples here but they have been clipped…

end

describe "Ruby to Ruby interactions" do

describe "when isolating Ruby classes" do

before do

@dagger = Dagger.new

@soldier = isolate Soldier

end

**it\_should\_behave\_like 'a ruby to ruby interaction'** 2

end

describe "when isolating Ruby classes with class members" do

before do

@dagger = Dagger.new

@soldier = isolate SoldierWithClassMembers

end

**it\_should\_behave\_like 'a ruby to ruby interaction'** 2

it "should work for an expectation on a class method without an argument constraint" do

@soldier.when\_class\_receives(:class\_name).return(5) 3

@soldier.class.class\_name.should == 5

@soldier.class.should have\_received(:class\_name)

end

end

describe "when isolating Ruby instances" do

before do

@dagger = Dagger.new

@soldier = isolate Soldier.new

end

**it\_should\_behave\_like 'a ruby to ruby interaction'** 2

it "should allow to delegate the method call to the real instance (partial mock)" do

@soldier.when\_receiving(:survive\_attack\_with).super\_after 4

result = @dagger.attack @soldier

result.should == 8

@soldier.should have\_received(:survive\_attack\_with)

end

it "should be able to isolate objects with constructor params" do

sheath = isolate Sheath

sheath.when\_receiving(:insert).raise("Already inserted")

lambda {

sheath.insert(@dagger)

}.should raise\_error(/^Already inserted$/) 5

end

it "should be able to isolate objects with constructor params" do

sheath = isolate(Sheath)

lambda { sheath.insert(@dagger) }.should\_not raise\_error

end

end

end

1. Define shared example group
2. Include shared example group
3. Mock class member
4. Define a partial mock
5. Override error raise

In the listing 9.12 we first define a shared example group (#1) and put all the common specs for the classes we’re about to test in there. We then include these shared examples in the actual context blocks for our examples (#2) by using the it\_should\_behave\_like method. This is a great way of practicing DRY in your tests. When we’ve got the shared examples out of the way we can focus on writing examples that are specific to our new use cases. The first one of the new use cases should answer the question: Can we mock class members (#3)? The answer when dealing with Ruby-to-Ruby interactions is: Yes! You can.

The examples for isolating a ruby instance answer 2 other questions for us. The first one is can we still forward method calls to the actual instance we’re mocking (#4). The next thing this instance isolation checks for is can we override error raises (#5). We can mock a fair bit of functionality now if we were to write only Ruby code we could probably cope with Caricature. But when you’re using IronRuby chances are that you’ll interact with CLR objects. The next part of this chapter deals with CLR to Ruby interactions.

9.3 Ruby talking to CLR objects

So far we’ve been talking strictly about testing ruby objects, this is about to end because we’ll start by integrating some CLR stuff into our specs. This time around we’re going to deal with classes that have been defined in a CLR language but will be consumed from the Ruby language.

In this scenario there isn’t anything that is especially different from the Ruby-to-Ruby interactions, this is because at this point we still have a ruby view on the world. So CLR objects will act like ruby objects when used in Ruby. That means that apart from the models (listing 9.13) not much has changed.

Listing 9.13: The CLR models

//C# portion of the models

using System;

using System.Collections.Generic;

namespace ClrModels {

public **interface** **IWeapon**{

int Attack(IWarrior warrior);

int Damage();

}

public **interface** **IWarrior**

{

int Id { get; }

string Name { get; set; }

bool IsKilledBy(IWeapon weapon);

int Attack(IWarrior target, IWeapon weapon);

int SurviveAttackWith(IWeapon weapon);

}

public **class** **Ninja** : **IWarrior**{

public Ninja(){

Name = "Tony the Ninja";

\_id = 1;

}

private readonly int \_id;

public string Name { get; set; }

public int Id { get { return \_id; } }

public int Attack(IWarrior target, IWeapon weapon){

return weapon.Attack(target);

}

public bool IsKilledBy(IWeapon weapon)

{

return weapon.Damage() > 3;

}

private int \_life = 10;

public **virtual** int SurviveAttackWith(IWeapon weapon){ A

return \_life - weapon.Damage();

}

}

}

# Ruby portion of the models

module InterfaceIncludingModule

include ClrModels::IWarrior

end

module InterfaceUpTheWazoo

include InterfaceIncludingModule

end

class InterfaceIncludingClass

include ClrModels::IWarrior B

attr\_reader :id

attr\_accessor :name

def is\_killed\_by(weapon)

end

def attack(target, weapon)

end

def survive\_attack\_with(weapon)

end

end

class SubClassingClrClass < ClrModels::Ninja

end

class InterfaceUpTheWazooClass

include InterfaceUpTheWazoo

end

A. mark method as virtual

B. define an interface on a ruby class

From the user’s (person writing the tests) point of view nothing changes really. Internally caricature will pick a different strategy for generating the proxy to the object. This time around it will use the reflection based API first to get all the methods defined on this class. When it’s an interface caricature needs to do some more inspections to get all the methods that could be defined so it knows how to swap those out with custom mocked implementations.

We’ll reuse those models and a few extra ones later on when we’re dealing with CLR-to-CLR interactions. Our models do have an interesting mix of inheritance stuffs and shows you how you would implement a CLR interface in a ruby class. From there it’s the same as in the CLR languages and you **have** **to** **implement all the methods defined on the interface**. These implementations can be empty stubs but nonetheless you have to define every method. What’s more you have to define **every method using the underscored name**! This rule isn’t reflected properly in the code listing because in this case it will actually be caricature that will take care of ensuring all the methods are there and that every method returns an appropriate default value for reference and value types.

Apart from the interface rule there are no differences when you would write a spec for a class like that. The full specs are included in the code samples for this chapter; you can find those in spec/integration/clr\_to\_ruby\_spec.rb. For completeness you can find 1 such spec below:

it "should work with an expectation with any arguments" do

@soldier = Soldier.new

@weapon = **isolate(ClrModels::IWeapon)**

@weapon.when\_receiving(:damage).return(5)

@soldier.is\_killed\_by?(@weapon).should be\_true

@weapon.should have\_received(:damage)

end

For the readers who haven’t dealt with a static language before, I’d like to warn you that confusion lies ahead. The symptoms the interface implementation exposes will get more complicated when you’re dealing with actual classes. And exactly this subject is discussed next.

9.4 Ruby as mediator between CLR classes

Up to this point we’ve been able to leave our happy go-lucky friend IronRuby in the illusion that it’s able to just charm its way through the execution of an application. Its toothpaste smile and smoothness have been what made our friend so popular. We’re about to burst that bubble though. As it turns out there are some places in .NET where he just can’t go without going through a makeover turning him from Fonzie into Richie. The first bits of the makeover are visibility rules for inheritance.

9.4.1 Visibility rules

The first thing that bit me when I started using IronRuby to develop a real application with mixed code in C# and IronRuby were the inheritance rules around visibility. These rules are different between Ruby and the CLR. Ruby has its own ideas when it comes to many things around programming, certainly so if you come from a CLR language background.

In C# you have to following visibility operators public, internal, protected, protected internal and private. Where public means visible to everybody and it screams: Please call me, please call me… The public is kind of like twitter for communicating. Where internal says: Ok, I guess I’m going to have to be called by people but I know so many people, I just want my friends to call me. A little like having an MSN account or a Jabber account where you have to authorize people first before they can send you messages. Then there is protected and that says: I’m only talking to my family; only my offspring can call these methods but I don’t care if my offspring lives in the same house (assembly) as I do. Which brings us to protected internal. Marking a method protected internal is the same as saying I’m only talking to my offspring or my friends (it’s an *inclusive* not an *exclusive* modifier). And the last one is private which means only the instance of that class has access, it is like going: This is mine, mine I tell you, all mine, my precioussssss...

Now Ruby has 3 visibility modifiers and they don’t exactly map to the ones you find in the CLR. Public means the same in both it’s the most permissive and everybody has access to that method. Ruby also has protected which is akin to internal on the CLR but instead of working on an assembly level it works on a class level. So instances of that class or its descendants can call the protected method like a like any other public method but instance of other classes can’t. Which brings us to private; private actually maps to protected on the CLR so no extra explanation needed here. If visibility in Ruby confuses you there is a good explanation available on the web by Jamis Buck <http://weblog.jamisbuck.org/2007/2/23/method-visibility-in-ruby>.

Apart from the visibility modifiers (public, internal, protected, protected internal and private), the CLR has more in stock for you. On the CLR you can also close classes completely (sealed), tell a class it’s missing its implementation (abstract) and you can make classes that can’t be instantiated (static). And at a method level you have one more modifier that you can use which tells the CLR that it’s allowed to override this property/method in subclasses (virtual). With the current implementation of caricature, sealed classes and static classes are problematic.

Why are static and sealed classes problematic?

The way caricature currently works is by subclassing a CLR class and then it uses reflection to create an object descriptor. From that descriptor it uses metaprogramming to implement the proxy for our object.

Now the CLR dictates that you cannot inherit off a sealed class nor can you inherit of a static class. You can see where the problems start. There are solutions around these problems but they currently aren’t implemented. Note that these facilities aren’t implemented in Rhino.Mocks or Moq either.

These rules have a big implication on how you can create isolations for classes that are bound by this ruleset, but are only in effect if you’re going to use Ruby as glue between classes defined in a CLR based language. This means that you’re CLR classes will need to have virtual methods for the ones you want to mock for usage in another CLR class. Table 9.2 shows an overview of the different modifiers and whether or not you can inherit of those.

|  |  |  |
| --- | --- | --- |
| C# modifier | Callable from outside | Inheriting a member |
| private | YES with –X:PrivateBinding | NO |
| protected | NO | HIDES OLD |
| protected virtual | NO | YES |
| protected internal | NO | HIDES OLD |
| protected internal virtual | NO | YES |
| internal | YES with –X:PrivateBinding | HIDES OLD with –X:PrivateBinding |
| internal virtual | YES with –X:PrivateBinding | YES with –X:PrivateBinding |
| public | YES | HIDES OLD |
| public virtual | YES | YES |

Table 9.2: The modifiers and their abilities

So why is this important, you could ask. In many cases the CLR won’t even know about the DLR or that the DLR has improved or changed the object. So when dealing with CLR objects it becomes important to understand how IronRuby generates its types.

When you define a ruby class then the DLR plays fully by the Ruby rules and these classes become a kind of named dictionary of method names and implementations (hugely simplified though). That means that when you make a method call the DLR looks it up in it’s dictionaries and then creates a call site, which will record the method call and execute the code.

However when you define a class that inherits of a CLR object or that includes a CLR interface then the DLR will actually generate a CLR type underneath. This CLR type gets generated in a different assembly, which is important to know for the internal modifier. So this generated type is a CLR type and as such is bound by CLR rules and that’s why you need to be aware of these inheritance rules.

In table 9.2 there is a HIDES OLD value. This means that the method will be declared as a new method hiding the method on the parent class. This is not the same as overriding because it’s not the implementation of the method that changes but it’s an entirely new method that starts life there and will be propagated to its subclasses. Using the new modifier in C# should be the last port of call because you could effectively break the Liskov Substitution Principle.

The IronRuby command-line takes a parameter –X:PrivateBinding, when your code runs in FullTrust, you can use this command-line parameter to gain access to private and internal members defined in the CLR assemblies you’re using from IronRuby.

Are all these rules a bad thing? Well they sure don’t make your life easier, however in many cases these libraries have been written with the idea that they will be used and extended with a CLR language so often the appropriate modifiers are in place. If it’s your own code you can easily provide the necessary hooks if it’s not your own code then it probably shouldn’t be you testing it anyway.

Is there a way around?

If you want more control over the types in the assembly you can look into the CLR Profiler API, which will let you intercept static methods a.o. Another option is to use Mono.Cecil, which allows you to do assembly rewriting so you can actually change the compiled code.

Ok example time. After this long theoretical explanation it might be good to see some of that in action. Listing 9.14 shows you some of the examples defined in spec/integration/clr\_to\_clr\_spec.rb.

Listing 9.14: Some specs for CLR interactions

describe "when isolating CLR instances" do

before do

@weapon = ClrModels::Sword.new

@ninja = Caricature::Isolation.for(ClrModels::Ninja.new)

end

it "should work without expectations" do

result = @weapon.attack @ninja

result.should == 0

@ninja.should have\_received(:survive\_attack\_with).with(@weapon)

end

it "should allow to delegate the method call to the real instance (partial mock)" do

@ninja.when\_receiving(:survive\_attack\_with).super\_after

result = @weapon.attack @ninja

result.should == 6

@ninja.should have\_received(:survive\_attack\_with)

end

end

Actually API wise there aren’t any changes, only now we’re a lot wiser about what happens under the covers and what we need to do in the C# classes. In listing 9.14 we test the interaction of the CLR classes Sword and Ninja. We also establish that we can still partial mock the CLR instance. We’re mocking the method survive\_attack\_with, we can do that because in listing 9.13 we marked the SurviveAttackWith method as virtual so we’re able to properly override it.

Ok I hope we have 1 pitfall avoided now, a few more lie ahead but this was the most complicated because both Ruby and C# have visibility modifiers, they just mean different things. The next port of call is events, a construct that doesn’t exist in IronRuby.

9.4.2 Handling Events

Whether you’re programming WPF, winforms, Silverlight or ASP.NET webforms you will be confronted with the event-handling infrastructure of the .NET framework. At its core the event handling from .NET is an Observer implementation, it has a collection in which you can store function pointers/delegates/closures. A class exposing an event (the observable) notifies any interested parties when some event has occurred by looping over the collection of delegates and invoking each one of them. Of course this is a simplification of what actually goes on but this way you get the drift of what they are. First stop: a refresher on handling events with C# and IronRuby.

Handling events

Most .NET languages have special constructs for defining events and working with them. For IronRuby they had to be piggy backed on. So before getting into testing events it might be worth it to just find out how to use events from IronRuby. Don’t worry it’s not hard, listing 9.15 shows code for a class with some events and a subscriber for this event in C#.

Listing 9.15: Handling events with C#

public class ExplodingCar {

public **virtual** event EventHandler<EventArgs> OnExploded; 1

public void Explode(){

// do logic here to make car explode

TriggerOnExploded();

}

protected **virtual** void TriggerOnExploded(){ 2

var handler = OnExploded;

if(handler != null){

handler.Invoke(this, EventArgs.Empty); 3

}

}

}

public class CleanupCrew : IDisposable{

private bool \_isDisposed = false;

private ExplodingCar \_car;

public CleanupCrew(ExplodingCar car){

\_car = car;

//don't use anonymous delegates or lambda's

**\_car.OnExploded += Handle\_carOnExploded**; 4

}

void Handle\_carOnExploded (object sender, EventArgs e)

{

// Do logic here when car exploded. Clean street, repair buildings etc.

}

public void Dispose(){

Dispose(true);

// Keep this here for subclasses that may use unmanaged resources

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool isDisposing){

if(!\_isDisposed){

if(isDisposing){

// cleanup managed resources here

// detach event handlers etc.

**\_car.OnExploded -= Handle\_carOnExploded**; 5

}

\_isDisposed = true;

//clean up unmanaged resources here.

}

}

}

1. define event
2. define trigger for event
3. execute event handlers
4. attach event handler
5. detach event handler

The code listing shows us a class ExplodingCar, which exposes an event OnExploded (#1). We define a method Explode where the logic lives to handle a car explosion. When that logic is done we call the method TriggerOnExploded (#2). This method checks if the event has actual handlers attached and if it has them attached it will invoke those event handlers (#3). That’s what is needed to expose an event, now to subscribe to the event we’ve defined the CleanupCrew class.

The CleanupCrew receives an instance of ExplodingCar and attaches an event handler Handle\_carOnExposed to the OnExploded event of the ExplodingCar instance (#4). Please note that you do want to clean up attached event handlers. While you certainly could use lambda’s or anymous delegates to attach an event handler you would still need to keep a reference around to this delegate so you can detach them properly later on. So to just go with the flow it’s easier to just use a method as event handler. Your colleagues will be thankful for that. Our CleanupCrew implements IDisposable and in the Dispose method we detach the event handler again (#5).

Why IDisposable?

In the C# example there is also an IDisposable implementation shown. That is an interface the CLR knows about and can be used to clean up unmanaged **and** managed resources. If you don’t want to create a bunch of memory leaks, you have to detach the event handlers again so the object can actually be garbage collected. A good example of such an application is a large or long-running WPF or SilverLight application, but basically any event-driven code is affected by this behavior. IDisposable provides a nice mechanism to implement that.

This goes for ruby too. Detach event handlers when you don’t need them anymore! The garbage collector is a great tool but you need to help it now and again.

Oh no! He’s there again with his C# stuff. I sure am, the reason for this is that because IronRuby is implemented in C# and because most of the .NET libraries you’ll use are all CLR based assemblies it’s of great help if you know those rules. Knowing how the C# looks and being able to translate that to working Ruby code will help you a lot when browsing the MSDN documentation for example. Listing 9.16 shows you how the CleanupCrew class would look if you were to implement that class in IronRuby.

Listing 9.16: the CleanupCrew in Ruby

class SwiftCleanupCrew

def initialize(car)

@car = car

@car\_exploded\_handler = method(:handle\_car\_on\_exploded) 1

@car.on\_exploded.add @car\_exploded\_handler 2

end

def handle\_car\_on\_exploded(sender, args)

# logic here to cleanup the street, repair buildings etc.

end

def dispose

unless @disposed

@car.on\_exploded.remove @car\_exploded\_handler 3

@disposed = true

end

end

alias :\_\_dispose\_\_ :dispose

end

1. Grab method
2. Attach event handler
3. Detach event handler

This implementation is a lot simpler than the C# version because at this point you have to inherit of a C# object to be able to use it as if it was a CLR object anyway, I’d suggest implementing the dispose pattern in the C# base class and then overriding the protected version of Dispose, because there is no way to keep the visibility in check. And if you have to inherit of a C# object anyway so you can use it as method parameters etc then it might be easier to create a DisposableObject base class. We’ve seen this dispose method earlier in Chapter 4 where it is being used in combination with a using method to mimic the C# using construct.

But I digress let’s get back to handling events. In listing 9.16 we create a car instance variable and grab the method, handle\_car\_on\_explode, that is going to function as event handler (#1). We then attach this unbound method to the on\_exploded event handler on the car instance (#2). Also in this class there is a dispose method where we detach the event handler from the on\_exploded event. The way we’ve just seen to add an event using the add method isn’t the only way to attach events. Below you find all the possible combinations to add an event handler to an object.

* @inst.on\_event.add @car\_exploded\_handler
* @inst.on\_event proc { |sender, args| handle\_event(sender, args) }
* @inst.on\_event.add proc { |sender, args| handle\_event(sender, args) }
* @inst.on\_event {|sender, args| handle\_event(sender, args) }

Now we know how to handle events, it might be a good idea to figure out how to test event handling with IronRuby.

Testing event-handling with IronRuby

Because we’ve been using Caricature to help us test objects, there is seemingly no difference between testing a Ruby object or a CLR object. Events, however, are a special case because in most CLR based languages there is a special syntax involved and it uses Delegates to get its job done. This is the reason why we’ll look at a full integration spec for event handling. We’d like to get the following output in the end:

CLR event handling for CLR interfaces

- should not raise an error when subscribing to an event

- should have 1 event subscription

- should raise an event when no block is given

- should raise an event with the provided parameters

- should allow overriding the default event handler

- should allow adding a block to the default event handler

- should verify if an event was raised

- should verify if an event was raised with specific parameters

CLR event handling for CLR classes

- should not raise an error when subscribing to an event

- should have 1 event subscription

- should raise an event when no block is given

- should raise an event with the provided parameters

- should allow overriding the default event handler

- should allow adding a block to the default event handler

- should verify if an event was raised

- should verify if an event was raised with specific parameters

The similarity in output is a good indicator that we’ll use a shared example to do this testing. We’ll also make use of shared contexts for the first time so we don’t have to define the same thing twice. The proposed output above actually proves that we can raise events and partially mock them too. Listing 9.17 shows the complete spec to get to the proposed output.

Listing 9.17: Testing event handling

require File.dirname(\_\_FILE\_\_) + "/../spec\_helper"

shared\_examples\_for "an event publisher" do

it "should have 1 event subscription" do

@proxy.isolation\_context.events.size.should == 1

end

it "should raise an event when no block is given" do

@proxy.when\_receiving(:explode)

.return(nil)

.raise\_event(:on\_is\_exposed\_changed) 2

@proxy.explode

@subscriber.counter.should == 1

end

it "should raise an event with the provided parameters" do

@proxy.when\_receiving(:explode)

.return(nil)

.raise\_event(:on\_is\_exposed\_changed, @sender, @ags)

@proxy.explode

@subscriber.sender.should == @sender

@subscriber.args.should == @ags

end

it "should allow overriding the default event handler" do

cnt, rsen, rar = 0, nil, nil

handler = lambda { |sen, ar| cnt +=1; rsen = sen; rar=ar }

@proxy.when\_receiving(:explode)

.return(nil)

.raise\_event(:on\_is\_exposed\_changed, @sender, @ags, &handler) 3

@proxy.explode

rsen.should == @sender

rar.should == @ags

cnt.should == 1

@subscriber.counter.should == 0

end

it "should allow adding a block to the default event handler" do

cnt = 0

handler = lambda { |sen, ar| cnt +=1 }

@proxy.when\_receiving(:explode) do |exp|

exp.return(nil)

exp.raise\_event(:on\_is\_exposed\_changed, @sender, @ags, &handler) 4

exp.raise\_subscriptions

end

@proxy.explode

cnt.should == 1

@subscriber.counter.should == 1

end

it "should verify if an event was raised" do

@proxy.when\_receiving(:explode)

.return(nil)

.raise\_event(:on\_is\_exposed\_changed)

@proxy.explode

@proxy.should have\_raised(:on\_is\_exposed\_changed) 5

end

it "should verify if an event was raised with specific parameters" do

@proxy.when\_receiving(:explode)

.return(nil)

.raise\_event(:on\_is\_exposed\_changed, @sender, @ags)

@proxy.explode

@proxy.should have\_raised(:on\_is\_exposed\_changed) do |ev|   
 ev.with(@sender, @ags) 6  
 end

end

end

describe "CLR event handling" do

before(:each) do

@sender = { :the => "sender" } 1

@ags = System::EventArgs.empty

end

describe "for CLR interfaces" do

before(:each) do

@proxy = isolate ClrModels::IExplodingWarrior

@subscriber = ClrModels::ExposedChangedSubscriber.new(@proxy)

end

it\_should\_behave\_like "an event publisher"

end

describe "for CLR classes" do

before(:each) do

@proxy = isolate ClrModels::ExposingWarrior

@subscriber = ClrModels::ExposedChangedSubscriber.new(@proxy)

end

it\_should\_behave\_like "an event publisher"

end

end

1. Use a shared context
2. Enlist event raise
3. Override event raise
4. Partially mock event raise
5. Verify event raise
6. Verify event raise and arguments

In listing 9.17 we declare a shared context with empty System::EventArgs and an object for the sender (#1). All the examples are defined in a shared example group and the first thing we check is if we can trigger raising an event (already defined in C#) from a mocked method (#2). The next example shows how we can override the C# event subscription with our own implementation just for the test (#3). When we know we can raise an already defined handler and provide our own handler we also check if we can do a combination of both (#4), which results in partially mocking an event raise.

So we know we can raise events but we’d probably also want to know if an event was actually raised. Caricature has a RSpec matcher for this and we use this matcher in the next example (#5). The first custom matcher doesn’t take arguments into account but the next example does (#6).

9.5 Summary

This chapter came disguised as a chapter about testing with IronRuby but as it turns out most of the pitfalls related with Testing have actually to do with CLR interoperability rules. Testing is a great place to find out what the shortcomings are of your library, and using IronRuby to do this testing makes your tests more succinct and perhaps more expressive too.

The fact that the Ruby community is on the front-line of improving testing of software, IronRuby has a leg up on his CLR brothers and sisters and gets really great testing libraries. We’ve seen a few of those libraries but then used one of the most popular ones to illustrate the possible pain points you’d run in to when testing with IronRuby.

We’ve seen that depending on whether your class inherits from a CLR object or includes a CLR interface IronRuby uses a different strategy to generate the functionality internally. When it has a CLR object somewhere IronRuby will resort to generating CLR types, but pure Ruby classes don’t have a CLR type as counterpart.

We’re aware of the shortcomings of Caricature, which have to do with CLR inheritance rules. We’ve also seen how we can deal with events defined on C# objects and how we can intercept them.

Armed with this knowledge you should be able to write most applications you write in C# today with IronRuby tomorrow. Our next chapter is a community contributed chapter which talks about transforming a Legacy Motif/Fortran77/napaBasic application to an IronRuby + WPF applcation and how Napa does this in an automated fashion.