.NET INTEROP

Getting Started With IronRuby And RSpec, Part 1

Ben Hall

This article is based on a prerelease version of IronRuby. All information is subject to change.

THIS ARTICLE DISCUSSES:

- Ruby and Duck Typing
- Ruby and the Microsoft .NET Framework
- Using IronRuby and RSpec

THIS ARTICLE USES THE FOLLOWING TECHNOLOGIES:

IronRuby

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY

Browse the Code Online

Contents

"That's not what we asked for!" I'm sure most developers have heard this cry from a customer shortly after delivering the latest build. The customer could be yelling at the poor developer for various reasons—maybe the requirements were not correctly understood or part of the system simply did not work.

Because customers are often unclear as to their own requirements, they will opt for the safety of the 100-page technical document that tries to define everything the system might have to do. Meanwhile, the developer struggles with undocumented legacy code, trying to understand how the app is intended to work while attempting to implement new requirements without breaking anything.

Times are changing. New approaches to software development are aimed at solving these problems, helping you to meet the customer requirements at the first attempt without causing failures in the process. These approaches are taking advantage of languages such as Ruby, allowing you to create easily readable and maintainable code with much shorter development iterations.

In this article, I will introduce you to Ruby and IronRuby and demonstrate some basics of Ruby interoperating with Microsoft .NET Framework-based code. I also explain how frameworks such as RSpec can be used to generate examples of how objects are intended to behave, providing both documentation and verification that the system is built correctly. This will set the stage for a future article in which I will explain acceptance testing in detail and demonstrate creating acceptance tests with IronRuby.

Defining Requirements and Examples in Code

To help write examples and define requirements you need a framework. There are many different approaches to writing automated acceptance tests or executable specifications. Some people use standard xUnit test frameworks successfully, while others use Fit and Fitness frameworks. I have found the best approach is to use Behavior-Driven Development (BDD). Dan North devised a BDD framework called JBehave as a way to define scenarios that describe the application's behavior in a way that can be communicated to the whole team, regardless of technical ability.

JBehave was the result of problems North faced with Test-Driven Development (TDD) and was implemented for the Java language. Later, North created RBehave, which has since been integrated into RSpec, a popular framework for Ruby. RSpec accommodates two different approaches to BDD: Dan North's approach, based on stories and scenarios to describe application's behavior, and Dave Astels' approach, which is more focused on creating examples at an object level.
C# does have some BDD frameworks, such as NSpec and NBehave. The main problem with writing tests in C# is that the true intent of the test is often hidden due to fact that you have extra structural elements and metadata such as braces, public and private. Overall, I don't think what is on offer via C# and NSpec/NBehave can match what is available via IronRuby and RSpec. Previously, this would have been a major problem for C# developers, as you could not have used the Ruby-based RSpec to test C# apps. With IronRuby, this is no longer a problem.
While still early in development, IronRuby is taking advantage of the Dynamic Language Runtime (DLR) to create an implementation of the Ruby language on top of the CLR. With IronRuby, you can employ existing Ruby apps and frameworks together with the .NET Framework and .NET-compliant languages. The result is that you can use the Ruby language and RSpec to test C# applications!
As a language, Ruby is concise, allowing you to write less code and express it in a much more natural manner, making your code easier to maintain. For example, to read all the lines of text from a file and write them out to the console, you would write this:
Copy Code
```
File.readlines('AboutMicrosoft.txt').map {|line| puts line}
```
The code opens the file AboutMicrosoft.txt and reads all the lines, passing each line into the block, with the line as a parameter that is then written to the console. In Ruby, a block is the collection of statements between the braces and is similar to invoking a method in C#, which uses the yield statement to return control to the calling method.
Its natural language approach is one of the reasons why Ruby is excellent to use when testing. The tests, or scenarios in this case, are much more readable.

Ruby and Duck Typing

One of the reasons why Ruby and dynamic languages are easier to read is due to how they handle typing. Instead of the developer defining a type, when the Ruby code is interrupted, the type of the variable is inferred. The language is still strongly typed, but it determines the variable's type dynamically.
With C#, interfaces allow for decoupling of objects while still defining a contract for the implementation. With Ruby, there is no need for contracts and interfaces. Instead, if the object has an implementation of the method you are calling, then it's called. If it doesn't, then an error is returned. As such, the contract for the object is defined by its implementation, not its relationship to other parts of the code.
To demonstrate this, I created a method that accepts a value, the type of which will be inferred at run time. It will then output Hello plus the value:
Copy Code
```
   def SayHello(val)
     puts "Hello #{val}"
   end
```
Because of the way types are handled, you can reuse the same method for both strings and integers without changing any of the code:
Copy Code
```
   SayHello("Test") => "Hello Test"
   SayHello(5) => "Hello 5"
```
I could have also have called the method without brackets, as these are optional in Ruby and can make the syntax much more readable:
Copy Code
```
   SayHello "Test" => "Hello Test"
```
This can be achieved in C# using object, but let's look at how this works with more complex objects. I defined a new method to output the result of a call to GetName. As long as the value of the parameter n implements a method called GetName, the code will work:
Copy Code

```
def outputName(n)
  puts n.GetName()
end
```

Here are two unrelated classes:

Copy Code

```
class Person
  attr_accessor :name
  def GetName()
    @name
  end
end
class Product
  def GetName()
    "The product with no name"
  end
end
```

Person has an accessor allowing the name variable to be set, while Product returns a hardcoded value. In Ruby, there is no need to write the return statement, the last result of the last line of code of a method is returned by default.

If you call the method outputName, the GetName method is called, demonstrating how Ruby's typing can enhance code reusability since we are not fixed to a single type:

Copy Code

```
outputName(Product.new) => "The product with no name"
$x = Person.new
$x.name = "Ben Hall"
outputName($x) => "Ben Hall"
```

If you call the method using an argument that doesn't implement GetName, then a NoMethodError is raised:

Copy Code

```
outputName("MyName") => :1:in 'outputName': \
  undefined method 'GetName' \
  for MyName:String (NoMethodError)
```

While this concept concerns some people, it can be useful; it allows for improved testability and application design, especially when testing C# applications, which I will discuss in the next section.

Ruby and MetaProgramming

Similar to other dynamic languages, Ruby embraces the concept of meta-programming. This allows you to extend any class or object at run time, enabling you to define methods and behavior at run time, giving you the ability to develop a self-modifying application. This is extremely useful in dynamic languages, especially in unit testing, as it allows you to customize the behavior of objects and methods to your own requirements.

In a similar fashion, you can extend the built-in Ruby classes to provide your own functionally, much like the extension methods in C# 3.0. With C#, there are various restrictions about the methods you can create. For example, they have to be static and in a separate class. This all harms readability. With Ruby, you just create a normal method on an existing class. For example, Integer is a class to handle whole numbers. Integer has a series of methods, but it does not have a method to say whether the number is even.

To create such a method, you simply create a new class with the same name (Integer) and define the new method. The question mark at the end of the method denotes that a Boolean is returned with the aim of making it easier to read:

Copy Code

```
class Integer
   def even?()
       self.abs % 2 == 0
   end
end
puts 2.even? => true
puts 1.even? => false
```

Ruby and the .NET Framework

In place of the old debate about whether to use a static or dynamic language, IronRuby lets you use the right language for the job. If it makes more sense to use C#, then you can use C#. If you need a more dynamic approach, you can easily use IronRuby, taking advantage of its interoperability with .NET and the dynamic nature of the Ruby language. I anticipate people using a mixture of different languages and technologies, such as C# for the main application and Ruby for testing.

The DLR provides the foundation to enable .NET interop. With this in place, you can take advantage of UI technology such as Windows Forms, Windows Presentation Foundation (WPF), and Silverlight while writing application code in Ruby.

Taking advantage of WPF from IronRuby is easy. When the following code is executed, you can have a fully functioning WPF window created from IronRuby. You still must reference mscorlib and the two WPF assemblies shipped with the .NET Framework 3.0, as you did with C#, using the require statements:

Copy Code

```
require 'mscorlib'
require 'PresentationFramework, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35'
require 'PresentationCore, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35'
```

You can now create and interact with the WPF objects. First, create aliases for all the objects you want to use. This makes the code cleaner, as you won't need to access the object via its namespace:

Copy Code

```
Window = System::Windows::Window
Application = System::Windows::Application
Button = System::Windows::Controls::Button
```

Next, create a WPF window and give it a title:

Copy Code

```
win = Window.new
win.title = 'IronRuby and WPF Interop'
```

Create a button in the same fashion:

Copy Code

```
mainButton = Button.new
mainButton.content = 'I'm a WPF button — press me'
```

In both instances I'm using the alias to access the full name of the object. When the button is clicked, I want to display a MessageBox to the user. As you would in C#, you can subscribe to the event and provide a block that is called when the click event is invoked:

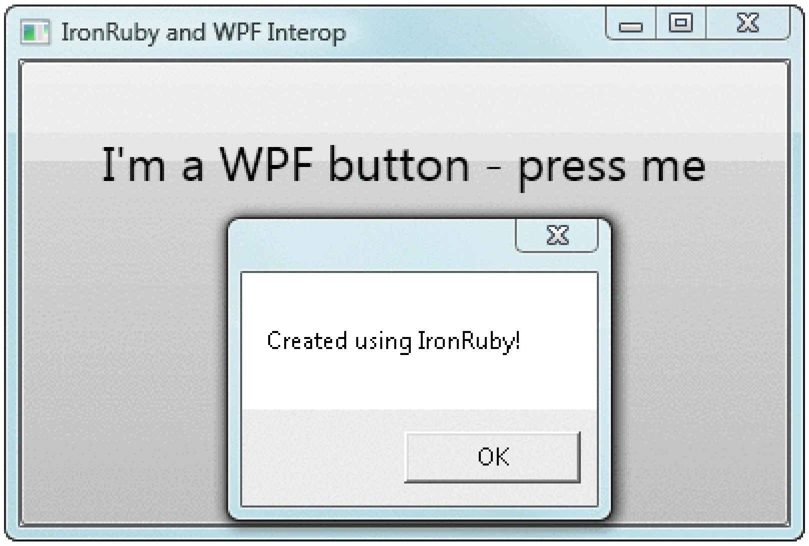
Copy Code

```
mainButton.click do |sender, args|
  System::Windows::MessageBox.Show("Created using IronRuby!")
end
```

Finally, set the content of the window to be the button, create a new Application object, and launch the window:

Copy Code

```
win.content = mainButton
my_app = Application.new
my_app.run win
```



Created using IronRuby!

Figure 1 **Using IronRuby to Create a WPF App**

At this point, a fully interactive WPF window will be displayed with the click event correctly wired up as shown in **Figure 1**. I interacted with the .NET Framework in the same way I would interact with Ruby libraries. One of the core principals set out by John Lam and team was to stay true to the Ruby language. This is an important principal for adoption by current Ruby developers, who won't want to change the way they create Ruby applications just because they're moving to IronRuby. Instead they can access all the rich .NET code in the same way.

Ruby and the CLR

Here is another example. If you attempt to access AddRange on IEnumerable, in C# the code would look like this:

Copy Code

```
ArrayList list = new ArrayList();
list.AddRange(new [] { 1,2,3,4 });
```

However, with Ruby, the accepted convention is for words within method names to be separated by underscores to improve readability. Creating a separate library to follow this convention would be too time-consuming and error-prone, and it would not support additional third-party development.

Instead, for CLR objects, IronRuby translates Ruby method calls into the equivalent method name for the CLR:

Copy Code

```
   $list = ArrayList.new
   $list.add_range([1,2,3,4])
```

Here I am accessing the AddRange method following Ruby's approach, which is lower case with words separated by an underscore. If you prefer, you can keep with the CLR naming convention as the method name still exists from Ruby:

Copy Code

```
$list.AddRange([1,2,3,4])
```

Both work the same; it's just a personal preference which to use.

As I mentioned, Ruby infers the type of objects at run time, and this is the same when handling C# objects. Consider a series of C# methods that return different object types, while the same object is being returned, the method signature returns either the interface or the concrete type. Within my example, I have an interface that defines a single HelloWorld method:

Copy Code

```
public interface IHello {
  string HelloWorld();
}
```

I created a Hello4Times class that inherits from Hello3Times, which inherits from the interface but has implemented three extra HelloWorld methods. Within the class I define a new method called HelloMethodOn4Times that calls the base implementation:

Copy Code
```
public class Hello4Times : Hello3Times {
  public string HelloMethodOn4Times () {
    return base.HelloWorld();
  }
}
```
I then defined a static class and methods that will return a new instance of the Hello4Times class but return it to the calling code as an interface. This means the calling code should only know about HelloWorld, not any of the additional methods defined:

Copy Code
```
public static class HelloWorld {
  public static IHello ReturnHello4TimesAsInterface() {
    return new Hello4Times();
  }
```

In my Ruby code, I have two method calls. The first call is on the interface-defined method, which you would expect to work without a problem. However, the second call is to the method on the concrete class. IronRuby has determined the type of the object being returned and can dispatch the method calls:

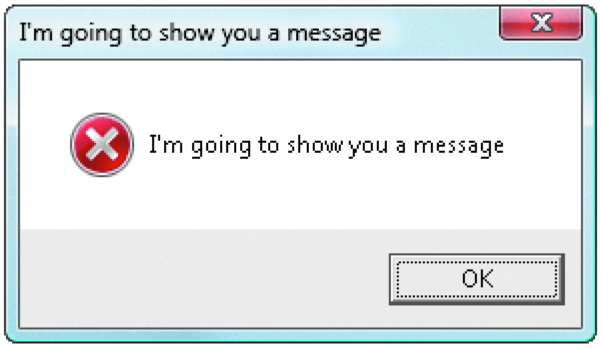
Copy Code
```
puts InteropSample::HelloWorld.ReturnHello4TimesAsInterface.HelloWorld
puts interopSample::HelloWorld.ReturnHello4TimesAsInterface.HelloMethodOn4Times
```
All of this happens without you ever needing to worry about casting the object to the correct type to call the method.

Following this theme, you can extend .NET objects in exactly the same fashion as you can with Ruby objects. When displaying a MessageBox, I don't want to keep defining which icons and buttons to use. Instead, I just want to provide a message.

Maybe you don't like the built-in functionality and want to extend the actual MessageBox class for seamless interaction. With Ruby, this is simple. You define a new class that is the same as the built in MessageBox within WPF. You then create a new method on the class that simply calls the show method with various defaults:

Copy Code
```
class System::Windows::MessageBox
  def self.ShowMessage(msg)
    System::Windows::MessageBox.Show(msg, msg, \
      System::Windows::MessageBoxButton.OK, \
      System::Windows::MessageBoxImage.Stop)
  end
end
```
After executing this block of code

Copy Code
```
System::Windows::MessageBox.ShowMessage( \
  "I'm going to show you a message")
```
you can call the ShowMessage method, the result being the message box shown in **Figure 2**.


I'm going to show you a message
I'm going to show you a message
OK
Figure 2 **Displaying a Custom MessageBox from Ruby**

Inside IronRuby

How is interop possible? The answer is the DLR. At a high level, when you execute Ruby code using IronRuby, a lot of work takes place behind the scenes. First, the code you write is tokenized and parsed by the IronRuby engine. The parsed code is then converted into the DLR's Abstract Syntax Tree (AST). This is a standardized AST that all language implementations, such as IronPython, need to provide to the DLR in order to execute the code.

Once the DLR has the AST, it converts the tree into Intermediate Language (IL). All .NET code is compiled down into IL to provide a generic language for the CLR. This is how IronRuby can interoperate with the .NET Framework—behind the scenes, all the code is executed as IL instructions. After the DLR has completed the conversion, it passes the IL to the CLR for execution and the result is returned to IronRuby.

During this process there are additional steps to improve performance and reliability, such as caching. For a deeper explanation, I recommend reading Bill Chiles's CLR Inside Out column "IronPython and the Dynamic Language Runtime" in the October 2007 issue of *MSDN Magazine*.

Within a separate assembly there is a hosting API that allows you to embed IronRuby within your own applications, allowing you to execute Ruby code from C# or users to execute their own code.

For more about the IronRuby implementation, download the entire source code from RubyForge. IronRuby is an open-source project implemented in C# and is an excellent example of a dynamic language implementation. IronPython is available as an open source project hosted at CodePlex, and it includes a sample language called ToyScript if you want an introduction into how the DLR works.

To ensure continued compatibility with the core Ruby platform, the IronRuby team is using RubySpecs. This is a shared set of examples based around how the Ruby language should be implemented. The aim of RubySpecs is to ensure different implementations including Matz's Ruby Interpreter (MRI), JRuby, MacRuby, and IronRuby have the same behavior. RubySpecs uses a syntax-compatible version of RSpec called MSpec. These are viewed as the acceptance tests for the IronRuby implementation.

Testing a C# App with IronRuby

For many years now, the awareness and practice of TDD has increased as a way to develop software, resulting in higher quality code, in terms of design and maintainability, along with fewer defects along the way. With IronRuby, I can use the RSpec specification framework and runner to provide examples of how my C# objects work, following a BDD approach instead of TDD.

While the scenario framework, which I'll cover in a future article, is more aligned with acceptance testing at the app level, the specification framework is more aligned to developers writing their own specifications about the code they are just about to implement and its expected behavior. The specification framework is based around providing examples describing the behavior at the object level. These examples can be run to verify that the implementation of the system is still working as the developer expects while providing documentation about how the objects are expected to behave.

This is an important distinction compared to unit-testing frameworks such as NUnit and MbUnit. Both of those use test attributes to indicate a method is a test for the system. RSpec takes a different approach. RSpec says that each method is an example of how the code is intended to work. While the distinction is subtle, it changes the way you write the examples, including the terminology you use, the way you organize the methods, and how easily the concept can be understood compared to TDD. With BDD and RSpec, together with the close integration of IronRuby and the .NET Framework, you can start using IronRuby to test C# applications. In terms of an example, the classic RSpec example is the Bowling game.

First, you need to access the bowling game's implementation in a C# assembly:

Copy Code

```
require File.dirname(__FILE__) + \
  '/InteropSamples/Bowling/Bowling/bin/Debug/bowling.dll'
```

You then need to access RSpec:

Copy Code

```
require 'rubygems'
require 'spec'
```

Now you can begin writing examples. This is an example showing how bowling implementation should work. RSpec has a Domain Specific Language (DSL) that you must follow for the examples to be executable. The first part of the DSL is the describe block. Here you simply state the object you are "describing" together with an optional description. In this case, I define the Bowling object, which will be implemented in C#:

Copy Code

```
describe Bowling, " defines the bowling game" do
```

To improve readability, any setup will be performed within a before block:

Copy Code

```
  before(:each) do
    @bowling = Bowling.new
  End
```

I am now in a position to interact with the object, create the example and verify that it works correctly. I use the "it" block, providing a string stating the context of the example and what it is demonstrating. I then write the section of code that interacts with the system and verifies the correct action has happened:

Copy Code

```
  it "should score 0 for a gutter game" do
    20.times { @bowling.hit(0) }
    @bowling.score.should == 0
  end
end
```

The C# implementation simply looks like this:

Copy Code

```
public class Bowling {
  public int Score { get; set; }
  public void Hit(int pins)
    { Score += pins; }
}
```

Finally, execute this to verify that the bowling example works as expected with RSpec testing C# objects:

Copy Code

```
>ir bowling_spec.rb
.
Finished in 1.0458315 seconds
1 example, 0 failures
```

If I wanted a more detailed report, I could select the format to be specdoc. This outputs the object description together with all the examples, stating if they have passed or not:

Copy Code

```
>ir bowling_spec.rb --format specdoc
Bowling defines the bowling game
- should score 0 for a gutter game
Finished in 1.43728 seconds
1 example, 0 failures
```

At the time of writing, the IronRuby team is not regularly shipping binaries. The team has said it is waiting until it is happy with the implementation, compatibility, and performance before releasing official binaries. However, because the source code is freely available, you can download the code and build it yourself.

To download the source code, you will need to have a Git source control client, such as msysgit, installed. The IronRuby source control is available online. If you would like more information then I recommend you visit the IronRuby project site or my blog post "Downloading IronRuby from GitHub." Once you have downloaded the source code, you can compile the assembly either with Visual Studio using the IronRuby.sln solution file or, if you have MRI installed, then you can use the command:

Copy Code
```
rake compile
```
Once you have IronRuby compiled, you must download various libraries, such as RSpec, to gain the extra functionality. Ruby has a concept of RubyGems where the gems are packages you can download to gain the functionality and the additional dependencies. To download RSpec, type the following at a command prompt:

Copy Code
```
gem install rspec
```
You would now be able to access the RSpec library from within your Ruby code. However, at the time of writing, RSpec does not work with IronRuby due to one or two bugs. Hopefully by the time this article is published RSpec should be working with IronRuby. To find out the status of the RSpec support, see the RSpec Web site or the IronRuby mailing list.

If the bug hasn't been fixed, I have made a binary with the bug fixed, together with the RSpec library in place. (Note: don't use this version once the team has fixed the problem.)

Once you have the correct binaries in place, ir.exe is the IronRuby interpreter that can execute your code. If you simply launch the app from the command line, you'll enter the interactive console. In Ruby, code is executed on a line-by-line basis, excellent for learning and debugging, but also for running short commands to solve daily problems. If you provide a file name as a parameter, then the file is executed with the results output to the console.

Moving Forward

In my next article, I will introduce the concept of acceptance testing and how it can improve communication between customer and developer. I will demonstrate how acceptance testing can be automated using IronRuby and RSpec to verify .NET applications and create an executable specification for the system.

Ben Hall is a C# developer/tester with a strong passion for software development and loves writing code. Ben works at Red Gate Software in the U.K. as a Test Engineer and enjoys exploring different ways of testing software, including both manual and automated testing, focusing on the best ways to test different types of applications. Ben is a C# MVP and maintains a blog at Blog.BenHall.me.uk.