.NET INTEROP

Automate Acceptance Testing With IronRuby

Ben Hall

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY

Browse the Code Online

THIS ARTICLE DISCUSSES:

- What is acceptance testing?
- Automated acceptance testing
- Implementing acceptance tests

THIS ARTICLE USES THE FOLLOWING TECHNOLOGIES:

IronRuby

Contents

Almost every developer dreams of the day when customer requirements can be correctly communicated without having to waste time implementing incomplete or incorrect functionality. Wouldn't it be great if we could hand a clear, readable specification to a customer and ask them if this matches their feature requirements, after which we could run exactly the same specification, without any additional effort, to verify our implementation against those requirements?

This level of communication between customer and developer can be successfully accomplished using the concept of acceptance testing and executable specifications. By taking advantage of Behavior Driven Development (BDD), we can start communicating requirements more effectively.

In my article in the February 2009 issue of *MSDN Magazine* ("Getting Started With IronRuby And RSpec, Part 1"), I introduced you to IronRuby and demonstrated how it lets you use the dynamic Ruby language to interoperate with .NET-compliant code such as C#. In this article, I will introduce you to the concept of acceptance testing. By building on the concepts introduced in my previous article, 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.

What Is Acceptance Testing?

Acceptance testing has been associated with many different definitions. For me, acceptance testing is more about verifying that the system under development meets the customer's requirements and less about reducing the number of bugs in our code. In other words, acceptance testing is not about testing code, but about building what the customer or business wants. It sounds obvious, but a lack of acceptance testing and a similar lack of understanding of the requirements are the main reasons for many project failures.

Acceptance testing only works with the support of the customer, or at the very least a proxy for the customer, to help define the criteria. Without someone driving the acceptance criteria, you have nothing to verify that the software is correct, which makes it difficult to verify you are building the correct software. The customer, together with all the members of the development team should come together to define the system in terms of a number of testable "scenarios" that describe what the system must do and how it must do it.

For example, if we were developing an e-commence system, an acceptance test might be based on shopping cart interaction. A typical scenario might be, "If the shopping cart is empty and I add Product 123, then the item count should increase to 1 and the subtotal should be $20." This provides an understanding of how the customer expects the shopping cart to behave and also provides some steps to verify the implementation.
As the system grows, you would have more scenarios verifying different parts of the system and feature set. Yet, an important consideration when writing the scenarios is the language used. The language should reflect how the business understands the problem, not how a developer would implement the solution. If the tests were described in terms of the actual implementation—"when I click the button 'Submit', the label 'Confirm' changes"—then this will provide less value to the customer and will be dependent on how you actually implement the system. If the actual implementation changes, but the business requirements remain the same, then this dependency will require additional maintenance costs as the team must update related tests.
By creating tests with clear requirements and passing criteria, your software stands a much better chance of meeting the customer's expectations. However, this still involves someone manually verifying that the requirements are met and the application works as expected. This is where the idea of automated acceptance tests comes in. Instead of the requirements being in an outdated document on a file share, the requirements are defined as examples and scenarios, are checked into source control with the implementation artifacts, and can be run at any point to verify whether any requirement is implemented and working properly. You can take the same approach for writing the tests, but instead of writing them in test case management software or in a spreadsheet, you write them directly in code

Automated Acceptance Testing

Acceptance testing will help to validate that you are building the application the customer wants, while automating these scenarios allows you to constantly verify the implementation throughout the development process and use them as part of your regression testing suite to ensure that future changes don't violate present requirements.
However, having a customer involved with writing tests, especially automated tests, introduces a number of potential problems. Customers, in general, are nontechnical and tend to shy away from the actual development of software. By collaborating with the technical team, the customer can provide the input and examples while testers or developers can quickly code the scenarios and executable specification. In addition, the scenarios must be clear to anyone within the business. By using IronRuby, you can improve the readability of tests, but how does this work in practice?
To illustrate how this process could potentially be used on a real project, I will demonstrate implementing one user story around a pricing calculation system. It's a simple example, but hopefully it will demonstrate the steps involved. Acceptance testing is a framework for how you can approach the problem, and there are many variants and interpretations. Nevertheless, this should provide you with the basic concepts allowing you to go forward and figure out which detailed techniques work for you.
This example came out of a "Red, Green, Refactor" presentation I gave at a local user group. Afterward, one of the attendees asked for my advice on how he could effectively unit test his application to ensure that all the price calculations, including various options and additions to the package, were being calculated correctly. This example is a perfect demonstration of where acceptance testing is extremely useful. By having a story and scenario around prices and what you expect to happen, you can be confident that the system is working correctly.
These verifications will also become a useful source of documentation for both the development team and the customers to demonstrate different examples of how the system works and the various exceptions. If a customer thinks that a price is incorrect, then the team can refer back to the executable specification, which the customer approved, as evidence of the work completed.
Without having these tests in place, the rules around pricing would likely be defined in huge Word documents attempting to explain how the system deals with various pricing configurations – documents that are inherently disconnected from the system being developed. This is why I believe in automated acceptance testing.

The Story

When developing a price calculation system with the aim of using acceptance tests, the first stage is to have everyone in a room define the story. Customers define what they want with encouragement from technical

members of the team. With BDD, as mentioned in the previous article, you want to use a set format for defining the stories in order to support a consistent language between members of the team, and specifically what is required from the customer. Given the following format, you would fill in the details with help from the customer:
As a [role] I want [feature] so that [outcome].
An immediate mistake is often to start including implementation details in the story. For example:
As an administrator, I want prices to be displayed in a grid view on default.aspx, so that I can provide them with the cost.
The story must describe the problem only in specific business terms and steer clear of technical details. An improved example could be this:
As a sales administrator, I want prices to be identified for a customer so that I can provide them with the cost.
Sadly, while it follows the format, it provides some very loose information, and it doesn't provide a starting point for the implementation. An even better example would be:
As a sales administrator, I want to be able to view prices for product x, so that I can provide customers with an accurate cost based on their requirements.
This story provides us with more detail and more context about what the customer is expecting to happen and what they actually want. As a result, it is much more useful to us when implementing the system.
One important fact to remember when writing stories and features is to keep them as focused as possible. It will make it much easier to create the scenarios, write the tests, and implement the finished code. In some systems, the story might be based around a feature.

Scenarios

Even with a story, you still need more information about what you are expecting to happen. Scenarios are extremely useful for this. A scenario simply states that given a certain context, if something happens, then the output should be this. The aim is to provide a more concrete example of how the story's implementation should behave in different situations. This communicates the expectations to the team and provides steps for verification.
The recommended syntax to use is known as the Given, When, Then (GWT) syntax, promoted by Dan North:
Given [context], When [something happens], Then [outcome].
An example scenario might be something like this:
Scenario: Single User License for Product X without support
Given Product X, When user requests single user license And this does not include support, Then the price should be $250.
The use of "And" can be used to connect multiple contexts or outcomes to allow you to give more meaning to the scenario and provide more information about what is happening.
Ideally, there should be as many scenarios as possible to resolve any ambiguity, serve as the executable specification, and provide enough information for the initial work to start. It's important to note that these scenarios are not fixed. As the work proceeds and more knowledge is gained, you might need to go back to the customer with questions about existing scenarios or create additional scenarios based on the knowledge gained.
When writing the scenarios, there are a few things to keep in mind. Similar to stories, scenarios should be focused on one example. The main reason for this is readability. If the scenario is doing too much, then the core message is lost.

Implementing Acceptance Tests

Once you have the story and scenarios in a clear, understandable format, the next stage is to automate the story and scenarios. This allows them to be run during development to track progress and catch regression bugs. While I will be talking in terms of RSpec, the fundamental process can be transferred to any BDD framework. For details on installing IronRuby and RSpec, see my previously mentioned article.
If you follow the previous example of writing your stories, then you will find the scenario runner in RSpec very natural. Within a plain empty file, in this case a file called pricing_scenarios_for_product_x, you simply copy the story and scenarios into the file in the following format:
Copy Code

```
Story: Pricing for New Product X
  As a sales administrator, I want to be able to view
  prices for product x so that I can provide customers
  with an accurate cost for their requirements.
Scenario: Single User License for Product X without support
  Given Product X
  When user requests a 1 user license
  And this does not include support
  Then the price should be $250
```

This now forms the basis of our acceptance criteria and will be used when verifying the application. However, to enable them to act as the executable specification, you need some Ruby code to execute them. IronRuby is an easy environment to start writing and executing code. Simply create a file with a .rb extension and begin writing RSpec tests.

The first step is to reference the RSpec framework. This is done via require directives. By including the following two lines, you can start using the RSpec story runner:

Copy Code

```
require 'rubygems'
require 'spec/story'
```

Now you can define steps. A step is a line within the scenario using the GWT format. With RSpec, a step is associated with a method in code. Each step should be associated with doing only one task. For example, the Given step is used for setting up objects, while Then is generally where the assertion and verification occurs. With RSpec, all the steps need to be wrapped in a block called steps_for. Inside the brackets is an identifier for that set of steps:

Copy Code

```
steps_for(:product_x) do
  #Steps go here
end
```

Each of the steps in the story directly relate to a line in the scenario. For example, the line "Given Product X" in the RSpec code would correspond to the following step method:

Copy Code

```
Given("Product $productName at $price") do |productName, price|
   pending "Need to complete implementation for accessing C# object"
 end
```

RSpec will perform string manipulation, allowing you to use placeholders in your step, with the placeholder value being set as a variable to the step. In this case, ProductName and Price will be stored in the variable, allowing you to reuse the same step for multiple different products and base prices.

Follow the same approach for the When and Then steps:

Copy Code

```
  When("user requests a $amount user license") do |amount|
    pending "Need to complete implementation for accessing C# object"
  end
  When("this does not include support") do
    pending "Need to complete implementation for accessing C# object"
  end
  Then("the price should be $price") do |price|
    pending "Need to complete implementation for accessing C# object"
  End
```

For the second step you don't need a placeholder because it is always the same action, therefore the block doesn't have a parameter. When you execute the tests, RSpec will call the correct methods in the correct order based on the scenario, replacing the values as defined.

Finally, you need to provide the path to the story file you created first. This is another block called with_steps_for, providing the identifier used when defining the steps. The body calls the run method with the path and name of the file where the scenarios are stored:

Copy Code

```
with_steps_for(:product_x) do
  run File.dirname(__FILE__) + "/pricing_scenarios_for_product_x"
```

```
end
```

To execute the tests, run the IronRuby command-line tool (ir), passing the Ruby file as a parameter:

Copy Code

```
>ir pricing_scenarios_for_product_x_story.rb
```

While writing the steps, you will notice that I was calling the pending method. This is to indicate that there is still work required to complete the functionality. As a result, when I run the tests, the output will state all of the pending tasks (see **Figure 1**). This is great for readability and understanding what is happening—the last thing you want is for the step to pass because there isn't any implementation.

Figure 1 Displaying Pending Methods

Copy Code

```
Running 1 scenarios
Story: Pricing for New Product X
 As a sales administrator, I want to be able to view prices for product x,
 so that I can provide customers with an accurate cost for their requirements.
 Scenario: Single User License for Product X without support
  Given Product X at 250 (PENDING)
  When user orders a 1 user license (PENDING)
  And this does not include support (PENDING)
  Then the price should be 250 (PENDING)
1 scenarios: 0 succeeded, 0 failed, 1 pending
Pending Steps:
1. Pricing for New Product X (Single User License for Product X without support):
Product $productName at $price
2. Pricing for New Product X (Single User License for Product X without support):
user orders a $amount user license
3. Pricing for New Product X (Single User License for Product X without support):
this does not include support
4. Pricing for New Product X (Single User License for Product X without support):
the price should be $price
```

RSpec Scenario Runner Interacting with C# Objects

At this point, the story, scenarios and steps are in place. However, we haven't implemented any code to interact with our system. In my previous article, I focused on how you can use the RSpec specification framework to provide examples of how my C# objects work and verify their implementation as an isolated unit. In this article, we are looking at how you can use the RSpec story runner for acceptance testing, focusing on verifying the complete application stack instead of isolated blocks.

The first task is to reference the C# assembly. IronRuby holds true to the Ruby language constructs and, as a result, references C# libraries the same as referencing Ruby libraries:

Copy Code

```
require File.dirname(__FILE__) +
  '/CSharpAssembly/CSharpAssembly/bin/Debug/CSharpAssembly.dll'
```

Now we can begin writing the body of our scenarios that we defined in the previous section. For the scenarios to pass, we need to implement the function for calculating the total price of an order. Within the Given block, you initialize the objects required for the scenario. For our scenarios, the only object required to be initialized at this point is the Product object. The constructor for the object requires the product name and the price. Both are obtained from the method parameter, which in turn is obtained from the scenario itself:

Copy Code

```
 Given("Product $productName at $price") do |productName, price|
   @product = CSharpNamespace::Product.new(productName, price.to_i)
 end
```

Once we have the initial objects, we need to set the variable state on those objects that will be the subject of our tests; this occurs in the When block. In our scenario, we state that a user has ordered a certain number of licenses, and our step needs to reflect this and set the object state accordingly:

Copy Code

```
 When("user orders a $amount user license") do |amount|
   @order = CSharpNamespace::Order.new(@product)
```

```
    @order.NumberOfLicenses = amount.to_i
  end
```

Finally, we come to the part where we actually verify that the above action worked correctly. Within our Then block, we define our expectations and assertions.

Here, we take the order we created and verify that the subtotal does match the price we defined within our scenario. Should is an extension method created dynamically by RSpec on all the objects allowing us to verify the truth of our statement; if it's not true, then an exception is raised:

Copy Code

```
  Then("the price should be $price") do |price|
    @order.Subtotal.should == price.to_i
  end
```

We can now execute the story by using the following command:

Copy Code

```
>ir pricing_scenarios_for_product_x_story.rb
```

When executing each scenario, RSpec will output the stories and scenarios to the console, if one of the steps fails, it will highlight the particular issue and indicate the total at the end:

Copy Code

```
Running 1 scenarios
Story: Pricing for New Product X
  As a sales administrator, I want to be able to view prices for product x,
  so that I can provide customers with an accurate cost for their requirements.

  Scenario: Single User License for Product X without support

    Given Product X at 250
    When user orders a 1 user license
    And this does not include support
    Then the price should be 250 (FAILED)

1 scenarios: 0 succeeded, 1 failed, 0 pending
```

If everything works successfully, then the following should appear on the command line:

Copy Code

```
Running 1 scenarios
Story: Pricing for New Product X
 As a sales administrator, I want to be able to view prices for product x,
 so that I can provide customers with an accurate cost for their requirements.
 Scenario: Single User License for Product X without support
  Given Product X at 250
  When user orders a 1 user license
  And this does not include support
  Then the price should be 250
1 scenarios: 1 succeeded, 0 failed, 0 pending
```

At this point we can start creating additional, more complex, scenarios to include more business logic. For example, you might want to include a discount on the price, with the scenario looking like this:

Copy Code

```
Scenario: Single User License for Product X with 20% discount
and includes 2 years unlimited premium support.
   Given Product X
   When user requests a 1 user license
   And including 2 years support
   And support length is unlimited
   And support type is premium
   And with 20% discount
   Then the price should be $800
```

As mentioned earlier, these scenarios use business terminology, making them readable not only to the customer but anyone in the business.

Acceptance Testing the UI

In my examples, the acceptance tests focused on the business logic and the domain objects in order to test whether the logic worked successfully. But what about how the user interacts with the application? These acceptance tests should be there to verify the logic is correct from the user's point of view—and the user's point of view is the UI.

Personally, I think acceptance tests should focus on the application logic and the important sections of the code that decide what to do. If your application has good decoupling and good separation of the logic from UI code, this should make the tests easier to implement. If you are testing at this level, your tests won't suffer from changes to the UI.

While the tests should focus solely on the logic, this doesn't mean you shouldn't have any acceptance tests around the UI. I like to have a set of smoke tests that target the core "happy path" of the UI. They focus on the parts of the application that users are most likely to use with the aim of getting the most value from the least amount of tests. If you attempt to cover all the possible paths and uses of the UI, and if the UI changes (such as options moving to different dialogs), then you would need to change all the tests.

For example, if you were testing the UI for an e-commerce site, the happy path would be to select an item, add it to your shopping cart, check out, and see the confirmation of purchase. If this scenario fails, you really want to know as soon as possible.

For certain applications, depending on complexity and lifetime, you might want to have more acceptance tests running on the UI to ensure that you have more confidence at the UI layer. Successfully testing the UI is a complex subject, though, and I don't have the space to cover it here. I recommend reading my blog posts on Project White for testing WPF and WatiN for testing Web apps. In addition, James McCaffrey has written on UI automation with Windows PowerShell in the December 2007 issue of *MSDN Magazine*.

Moving Forward

At this point, you are ready to start putting IronRuby to work on your acceptance tests. The information here should get you started, but there are some things you should keep in mind.

One possible issue is that IronRuby isn't ready for production use just yet. While you could use it today, there is continuing development and refinement of both IronRuby and RSpec. That means bug fixes, but also the possibility of breaking changes. There are also issues around the speed of execution of RSpec. However, it's worth evaluating whether the issues outweigh the advantages you get in simplifying your tests.

Another concern that some people may hold relates to context switching. Generally, people like to write tests and production code in the same language. While I generally agree, for unit tests where you are doing a large number of switches on a daily basis, you might not want to use IronRuby and RSpec. However, for acceptance testing, the amount of context switching needed is much lower. As a result, I think you can justify the context switching in order to take advantage of the readability and the more natural way of writing the verifications and scenarios because they will provide huge benefits to the customer and development team.

In the end, integrating acceptance testing into the development process can be a hugely positive step for a development organization. By using techniques such as stories and scenarios to define functionality in customer-oriented terms, coupled with readable yet automated acceptance tests, you will be able to create a more trusted partnership with your customer and consistently deliver more robust and reliable software.

Ben Hall is a UK C# developer/tester with a strong passion for software development. He loves writing code. Ben works at Red Gate Software 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