8

Sugar-coating the CLR with IronRuby

This chapter covers:

* Building DSL’s with Ruby
* Leveraging metaprogramming

In my dealings with other programmers I make it no secret how I feel about the ruby language. One of the aspects that appeal most to me is the malleability of the language. You can make it do a lot without the need for complicated parsers. So one of the things that are very easy to do in ruby is create internal domain specific languages (DSL’s) for a bunch of tasks, one such DSL is RSpec, which is used for Behavior Driven Development (BDD). And this chapter is going to explore some of the things you can create DSL’s for and more importantly how you can go about it. This chapter is by no means a complete explanation on the subject about DSL’s but offers a general introduction.

If you want to read a good book on building DSL’s I can recommend Ayende’s book, Building Domain Specific Languages in BOO (http://www.manning.com/rahien/), who does an excellent job at explaining the intricacies of building these tools with boo.

The original CLR languages are classes designed to use in a static language and perhaps even with intellisense in the back of the programmers mind. Needless to say the API’s don’t always flow as nicely as you’d like to and often don’t express what you want the computer to do as clearly as could be. Or you just want to improve on some bits and pieces of the API in those cases and many more a DSL might be exactly what you need. In this chapter we’ll integrate an existing DSL as an external DSL to C#. We’ll create an internal DSL to watch the file system for changes. As usual we’re first going to start with a minimum of background information about DSL’s.

8.1. What? Why? Who?

I think most of us have been there, you were solving a problem; it was a tough nut to crack but eventually you did. The problem you took so many approaches you’re code has looked better and if the next guy was an axe wielding psychopath, you’d probably just have enough time to arrange your own funeral. Luckily the axe-wielding psychopath is you in this case so that won’t happen but as Murphy’s law prescribes, there are a bunch of things that come up and more important tasks need to be handled first. Anyway I think you know where this is heading the time between writing the code and getting back into is long enough for it to not be fresh in the back of your head anymore and the code you wrote isn’t obvious when you glance at it. This is where DSL’s come in; the purpose is to make the intent of the programmer clearer to the person maintaining the code than to the computer.

There are a couple of variations of DSL’s and we’ll talk about the internal and external variants, but not about the visual DSL’s that seem to be popular in some circles or fluent interfaces which are really just a poor mans (language) internal DSL. I for one am still not convinced about the value of visual DSL’s as I seem to fight them more than get on with my job, I also feel very much out of the driver seat when using those. If all goes well and everything works then great but this is rarely the case. That could have more to do with me than with the actual tool. To me a programmer works at an abstract level where a designer works visually.

Ok that leaves us with the question so what’s the difference then between an internal and an external DSL. An internal DSL is one you craft with the same language that will consume it. Another name for an internal DSL would be embedded DSL and some examples within the Ruby language are Rake, RSpec, builder,… An external DSL is a DSL you or somebody else has built from scratch, often using a lexer and a parser to create an AST which can then be executed by the DSL host, examples of external DSL’s are SQL, FIT, configuration files, …

Good now that it’s established that all of us use DSL’s on an almost daily basis let’s start by the simplest case of just integrating a ruby DSL into a C# application.

8.2. A state machine integration

When I was preparing for this chapter, which was a direct consequence of some of the feedback I got from the reviewers, I asked the IronRuby community if they have .NET CLR a project they’d like to see ported to IronRuby as an example. I received a few emails and the project that stood out the most to me was the stateless project from Nicholas Blumhardt (http://code.google.com/p/stateless/), who also produced the excellent Autofac IoC container. After looking into that project a little bit closer it became clear that it was a port of a project on codeplex that uses boo as the language to create the DSL. Googling some more brings me to the statemachine gem, which sure looks like a really good match to both of the aforementioned projects.

At this point the plan changes and I’m going from: Okay I’m going to build a statemachine DSL to wouldn’t it be interesting to just make this DSL usable from C# somehow, so lets explore that a little, starting with the end result. I also want to make it clear that the codes and explanations provided are here to highlight the IronRuby with C# interaction, they are in no way criticism on the original C# example code.

8.2.1 The end goal of this exercise

At the end of this part in the book we should be able to run the Bug sample from the stateless project as shown in listing 8.1. It creates an instance of a C# bug class and moves that through a few states before closing the bug. We should be able to just take this Main method and replace the bug class with our implementation and it should still work.

Listing 8.1: The original stateless sample

class Program

{

static void Main(string[] args)

{

var bug = new Bug("Incorrect stock count");

bug.Assign("Joe");

bug.Defer();

bug.Assign("Harry");

bug.Assign("Fred");

bug.Close();

Console.ReadKey(false);

}

}

If we manage that I guess the application working with the Bug class will still be happy as it doesn’t know bug has changed internally to be very much improved, after all it will be using ruby for its stuff that’s an improvement in itself. The code listing above prints out the following messages to the screen:

+ivan@ivan-mbp:~/projects/gitosis/stateless/BugTrackerExample/bin/Debug

(master)» mono BugTrackerExample.exe

Joe, RE Incorrect stock count: You own it.

Joe, RE Incorrect stock count: You're off the hook.

Harry, RE Incorrect stock count: You own it.

Harry, RE Incorrect stock count: You're off the hook.

Harry, RE Incorrect stock count: Don't forget to help the new guy.

Fred, RE Incorrect stock count: You own it.

Fred, RE Incorrect stock count: You're off the hook.

So we get some output that shows the states and transitions with output the bug instance went through. We’re ready to begin converting the code from the stateless project to a ruby-enabled version.

8.2.2 Setting the stage for the example

As it turns out the statemachine gem uses a context to actually execute the application code so this example is going to be mostly about a quick demo for hosting this stuff in an existing application. So lets dive straight in with an example. The example, however, is a port of the Bug example of the stateless project, the C# version of the definition we’ll create in ruby is shown in listing 8.2.

Listing 8.2: The C# version of the statemachine definition

public class Bug

{

enum State { Open, Assigned, Deferred, Resolved, Closed }

enum Trigger { Assign, Defer, Resolve, Close }

State \_state = State.Open;

StateMachine<State, Trigger> \_machine;

StateMachine<State, Trigger>.TriggerWithParameters<string> \_assignTrigger;

string \_title;

string \_assignee;

public Bug(string title)

{

\_title = title;

\_machine = new StateMachine<State, Trigger>(() => \_state, s => \_state = s);

\_assignTrigger = \_machine.SetTriggerParameters<string>(Trigger.Assign);

\_machine.Configure(State.Open)

.Permit(Trigger.Assign, State.Assigned);

\_machine.Configure(State.Assigned)

.SubstateOf(State.Open)

.OnEntryFrom(\_assignTrigger, assignee => OnAssigned(assignee))

.PermitReentry(Trigger.Assign)

.Permit(Trigger.Close, State.Closed)

.Permit(Trigger.Defer, State.Deferred)

.OnExit(() => OnDeassigned());

\_machine.Configure(State.Deferred)

.OnEntry(() => \_assignee = null)

.Permit(Trigger.Assign, State.Assigned);

}

public void Close()

{

\_machine.Fire(Trigger.Close);

}

public void Assign(string assignee)

{

\_machine.Fire(\_assignTrigger, assignee);

}

public bool CanAssign

{

get

{

Console.WriteLine ("in Can Assign");

return \_machine.CanFire(Trigger.Assign);

}

}

public void Defer()

{

\_machine.Fire(Trigger.Defer);

}

void OnAssigned(string assignee)

{

if (\_assignee != null && assignee != \_assignee)

SendEmailToAssignee("Don't forget to help the new guy.");

\_assignee = assignee;

SendEmailToAssignee("You own it.");

}

void OnDeassigned()

{

SendEmailToAssignee("You're off the hook.");

}

void SendEmailToAssignee(string message)

{

Console.WriteLine("{0}, RE {1}: {2}", \_assignee, \_title, message);

}

}

I’m not going to explain the C# code above but I’ve included it to have something to contrast against later on. The first thing we need to do is make sure we’ve got the statemachine gem installed, by executing igem install statemachine.

» igem install statemachine

Successfully installed statemachine-0.4.1

1 gem installed

Installing ri documentation for statemachine-0.4.1...

Installing RDoc documentation for statemachine-0.4.1...

Now that we have the statemachine gem it might be a good idea to open its documentation so you can look up something if you need it.

Getting to the docs easily

The biggest part of the gems are distributed as ruby source files and most of them include documentation. That means you have all the information you need on your machine locally to figure out how to use certain libraries. In many cases the unit tests are also a part of the library and sometimes there is an examples directory to get you on your way.

Now how do you get to that documentation which is provided as html. There are 2 tools ruby provides for this. The first one is ri and you can interrogate it from the command line provided that the docs are generated beforehand.

You can generate the missing docs with the gem command rdoc. So for example for caricature you can do igem rdoc caricature and that should then generate the docs for the caricature gem.

To search for the documentation on the String class from the command-line, you would enter the command iri string. And that would then show you all the classes on your system that have a method with the word string in them and it allows you to drill down to the information you need.

While it is cool you get that kind of access to docs. I will generally use a web browser to go through the API documentation and I have a gems server running which serves me all those RDoc documentation files.

To start that server do igem server and afterwards you can open a brower and point it to <http://localhost:8808/> This will show you an overview of all the gems you have installed on your system and allows you to quickly go to the gem you want the documentation for.

This gets our ironruby set up correctly for the next steps and I’ll try to show you a typical workflow when building a ruby application. Next up: create the C# project.

8.2.3 Creating the C# project

The next thing to do is to open visual studio/monodevelop and create a new C# console application, I called mine BugTrackerStatemachine. Next we’ll have to add the references to the IronRuby libraries and then we’re ready to commence building the state machine. Figure 8.1 has the screenshot of the add reference dialog box on visual studio. I’ve extracted the zip file for IronRuby to C:\ironruby so this means the binaries are located in C:\ironruby\bin.

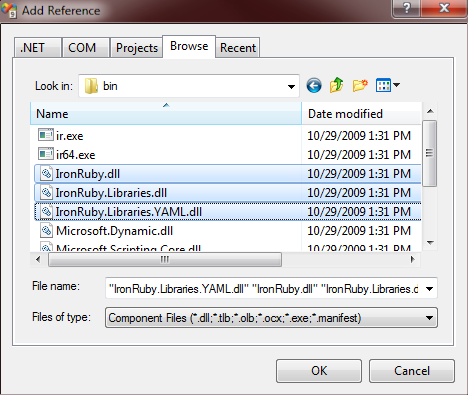


Figure 8.1 Add Ironruby references to the project

This particular example can be run on mono too so figure 8.2 shows the same dialog box but for monodevelop. I’ve extracted the ironuby zip file to my home directory so the binaries are located in ~/bin.

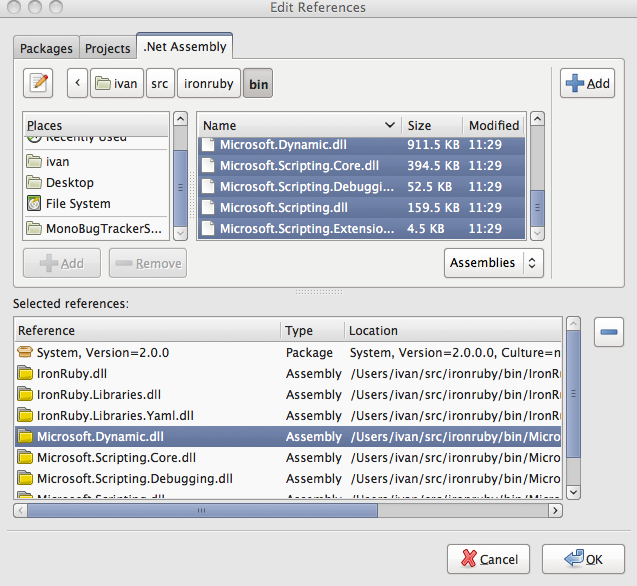


Figure 8.2: The MonoDevelop edit references dialog box

If you’ve followed along earlier and started the gem server then you can visit this url <http://localhost:8808/doc_root/statemachine-0.4.1/rdoc/classes/Statemachine/StatemachineBuilder.html> and that shows us that the statemachine builder takes a context, we’re going to take advantage of this fact to provide it with a C# context for this example. In this case I’ve chosen to use the context as the integration point because it was the most convenient and whenever something seems to fit I’ll try that first instead of swimming upstream.

8.2.4 Creating the definition in ruby

One might ask what’s the point of all this? We already have a perfectly good working solution in C# so why would I want to do this from with IronRuby. There are a number of reasons. While I can read the code from listing 8.2, it’s not as clear as it could be and it requires me to actually decipher the states in the constructor. If I want to make a change to the workflow I have to recompile my application while with the Ruby version you are able to change it at even run-time.

Tip:

You can use Ruby instead of XML for configuration tasks and not induce carpal tunnel syndrome for your fellow devs. It gives you the benefit that you can work with real types and you can properly unit test it while it’s still very much friendly for machines it will make a world of difference to your developers.

Another consequence of what we’re going to do is to separate the application logic from the infrastructural code so it ultimately results in a better separation of concerns, keep in mind that we are working with sample code here though. To be able to port the fluent interface shown in listing 8.2 let’s first determine what we don’t need. We won’t have to worry about enum values they are internal values for the state machine. Then, there is a bunch of noise to declare instance variables/fields with some generics, which we won’t need those either. Then there are 2 instance fields for the assignee and the title. That sounds pretty useful lets keep those but let’s replace them with public getters as that reflects a more useful bug object. So in our own C# application we now have a class that looks like listing 8.3.

Listing 8.3: including the necessary properties

public class Bug

{

public string Title { get; private set; }

public string Assignee { get; private set; }

public object Statemachine { get; set; } 1

}

1. the Statemachine property

The observant reader, as I’m sure you all are, will have noticed that there is a Statemachine property with a getter and setter on this class. This is a result from what I’ve read in the documentation of the statemachine gem. The following link shows the source of the context method on the statemachine builder and it sets a statemachine property: <http://localhost:8808/doc_root/statemachine-0.4.1/rdoc/classes/Statemachine/StatemachineBuilder.src/M000017.html>. So I’ve added that one too. We’ll also need a path to the ruby file but’s lets worry about that a little bit later and jump into the constructor where the statemachine is being defined. From that code we can derive that there are 4 states (Open, Assigned, Deferred and Closed) and 1 super state (Open). We can also see that from the 4 transitions (triggers) defined in that sample that there are only 3 in use (Assign, Defer and Close). This gives us a pretty good idea of what we’re going to put in our ruby state machine definition. The definition in listing 8.2 shows a few more items like the fact that an Assigned state has an OnEntryFrom and OnExit callbacks and the Deferred state has an OnEntry callback. Listing 8.4 shows the ruby equivalent of the C# state machine definition.

Listing 8.4: The state machine in Ruby

require 'rubygems'

require 'statemachine'

**inserted = ctxt** 6

Statemachine.build do

trans :open, :assign, :assigned, :on\_assigned 1

trans :assigned, :close, :closed

trans :assigned, :defer, :deferred

superstate :open do 2

state :assigned do

event :assign, :assigned, :on\_assigned 3

on\_exit :deassigned 4

end

end

state :deferred do

event :assign, :assigned, :on\_assigned

on\_entry :deassign 5

end

context inserted 7

end

1. transition and action

2. super state declaration

3. formal transition definition

4. on exit action

5. on entry action

6. bringing in the variable from the DLR scope

7. setting the context

This book is not about explaining the intrinsics of state machines but Micah the author of the statemachine gem does a pretty good job:

<http://blog.8thlight.com/articles/2006/11/17/understanding-statemachines-part-1-states-and-transitions>

<http://blog.8thlight.com/articles/2006/11/30/understanding-statemachines-part-2-actions>

<http://blog.8thlight.com/articles/2007/02/13/understanding-statemachines-part-3-conditional-logic>

<http://blog.8thlight.com/articles/2007/04/07/understanding-statemachines-part-4-superstates>

The statemachine gem takes the approach that a statemachine is defined by its transitions so we first define a transition (#1) for going from open to assigned and it invokes the action on\_assigned. This action will be invoked on the context (#7), to which we’ll return later. We define the transitions from the assigned state and proceed with creating a super state open (#2). This super state is the SubStateOf method in listing 8.2. Inside the super state we define a state assigned and this state has a reentry transition defined with the formal transition syntax (#3). The assigned state also defines an on\_exit action (#4). This action will also be invoked on the context (#7).

We move on to defining the deferred state. This state defines a transition back to assigned and it also defines an on\_entry action (#5). The implementations of this behavior will reside in our C# bug class. So the last thing left to explain is how we’re going to link the C# bug class with our Ruby state machine. The bolded text takes care of that and injects the ctxt variable into the scope by assigning it to the inserted variable (#6). We assign the inserted variable as context inside the Statemachine.build method (#7). By assigning it there it also checks for the statemachine setter on the context and if present it will set the statemachine to the current instance.

Why is there inserted = ctxt ?

That is because a variable you set with ScriptScope.SetVariable (later in this chapter) isn’t a ruby variable but rather a DLR variable. Here’s what Tomas Matousek, a Microsoft SDE on IronRuby, had to say on the subject:

**inserted = ctxt (#6)** calls a “ctxt” method on “self”. The hosting API injects method\_missing into the top-level singleton – you can see it by printing p self.method(:method\_missing). The implementation of that method\_missing looks into the scope for variables.

In the latter case, I assume, you instance\_eval the block against some object, right? Hence ctxt is an invocation on that object, which has no relationship with the scope.

This is completely correct. The Statemachine.build method actually really does use instance\_eval and this changes the context of the block to the instance that is executing the block. So all method calls are evaluated on the Statemachine instance instead of in the global scope. And that is why we have that line (#6) there.

At first I thought that setting a scope variable would be the same as declaring a variable in ruby but the 2 don’t match up completely.

That takes care of the ruby definition of the state machine and we’re ready to go back to our C# class to finish the integration. The first thing we’re going to do is host the ironruby runtime.

8.2.5 Hosting the IronRuby runtime

We’re moving back into C# now, to finish up our bug implementation, when we left it back in listing 8.3 our class had 3 properties but we’re going to need a few more. We will need the path to the definitions file so it can be overridden so we’ll make that a read/write property (#1) as shown in listing 8.5.

Listing 8.5: Instrumenting the Bug class with IronRuby

public class Bug

{

public string Title { get; private set; }

public string Assignee { get; private set; }

public object Statemachine { get; set; }

public string DefinitionPath { get; set; } 1

private ObjectOperations \_rubyOperations;

private static ScriptEngine \_engine = Ruby.CreateEngine(); 2

public Bug(string title)

{

DefinitionPath = Path.Combine(

Environment.CurrentDirectory, "definition.rb"

);

Title = title;

ExecuteRubyDefinition();

}

private void ExecuteRubyDefinition(){

\_rubyOperations = \_engine.CreateOperations(); 3

var scope = \_engine.CreateScope(); 4

scope.SetVariable("ctxt", this); 5

\_engine.ExecuteFile(DefinitionPath, scope); 6

}

}

1. The defintion path property
2. Creating the ruby engine
3. Getting the object operations helper
4. Creating a new scope
5. Setting a scope variable
6. Execute the state machine definition

We’re also going to need a ScriptEngine (lives in Microsoft.Scripting.Hosting), because these engines are pretty expensive to boot so I made it a static field on the bug class (#2). This way the engine gets booted when the type is initialized (not the instance) and only once during the run of the application. So if you have more classes that are going to use IronRuby it might pay not to have too many of those around and share them as much as possible, depending on your requirements.

We’re now entering the constructor and we initialize the definition path with a default path. We also initialize the Title property with the value from the constructor parameter and proceed with executing the ruby definition.

The first thing to notice is that we ask the script engine to create an object operations helper (#3). We’ll need this to invoke methods on our ruby state machine later on to replace the methods that have a call to \_machine.Trigger from the Bug class in listing 8.2. Next we create a scope (#4) so we can set variables for that scope before executing the file. We then set a variable ctxt to this instance of the bug in this scope (#5). And the last thing we do is we execute the file in the same scope (#6) so that it has access to the scope variable we set previously. This ctxt variable is then used to initialize the inserted variable in listing 8.4. And this brings us to the part where we have to implement the behavior (actions) on the Bug class.

8.2.6 Implementing the behavior on the Bug class

To implement the behavior on the class we can basically copy/paste the methods OnAssigned and OnDeassigned from the original example in listing 8.2 and change their visibility to public. This is shown in listing 8.6, the changes to these methods from the once you can find in listing 8.2 have been marked in bold.

8.6 Adding the actions to the bug class

**public virtual void Deassign(){**

**Assignee = null;**

**}**

public virtual void OnAssigned(string assignee)

{

if (Assignee != null && assignee != Assignee)

SendEmailToAssignee("Don't forget to help the new guy.");

Assignee = assignee;

SendEmailToAssignee("You own it.");

}

public virtual void Deassigned()

{

**if(Assignee != null)** SendEmailToAssignee("You're off the hook.");

}

public virtual void SendEmailToAssignee(string message)

{

Console.WriteLine("{0}, RE {1}: {2}", Assignee, Title, message);

}

The first thing that is different from the original listing is that we added a Deassign method. This just made it easier and more consistent for this chapter. The other change I made was checking if there is an Assignee before we alert them that they’re off the hook. This is because the stateless implementation and the ruby gem state machine behave slightly different. These methods implement the actions we use in listing 8.4, and we’re getting close to finishing this all we need to do now is make sure we can navigate between the states and to do that we’re going to use the object operations helper we created earlier on in listing 8.5. The transition actions (assign, defer, close) are created on the ruby state machine instance so we need to invoke some methods on that object and pass it a parameter as shown in listing 8.7.

Listing 8.7: Invoking members on the ruby class from C#

public virtual void Assign(string assignee){

\_rubyOperations.InvokeMember(Statemachine, "assign", assignee); 2

}

public virtual void Defer(){

\_rubyOperations.InvokeMember(Statemachine, "defer"); 1

}

public virtual void Close(){

\_rubyOperations.InvokeMember(Statemachine, "close");

}

1. Invoke a method on an object
2. Invoke a method with parameters

The first method from the above code listing we’re going to discuss is the Defer method. It uses the \_rubyOperations.InvokeMember (#1) method to execute a method on the ruby class. The ObjectOperations helper for the DLR has all kinds of goodness for interacting with dynamic language objects. It contains methods to interrogate those objects as well as methods to invoke members and manipulate the objects by adding methods or variables or … The second example (#2) shows invoking the assign method with a parameter. Listing 8.9 shows the completed listing for the Bug class. It’s only a little bit shorter than the C# version so there is no win in LOC there is a win in readability though.

Listing 8.9: the completed listing

public class Bug

{

public string Title { get; private set; }

public string Assignee { get; private set; }

public object Statemachine { get; set; }

public string DefinitionPath { get; set; }

private ObjectOperations \_rubyOperations;

private static ScriptEngine \_engine = Ruby.CreateEngine();

public Bug(string title)

{

DefinitionPath = Path.Combine(

Environment.CurrentDirectory, "definition.rb"

);

Title = title;

ExecuteRubyDefinition();

}

private void ExecuteRubyDefinition(){

\_rubyOperations = \_engine.CreateOperations();

var scope = \_engine.CreateScope();

scope.SetVariable("ctxt", this);

\_engine.ExecuteFile(DefinitionPath, scope);

}

public virtual void Deassign(){

Assignee = null;

}

public virtual void OnAssigned(string assignee)

{

if (Assignee != null && assignee != Assignee)

SendEmailToAssignee("Don't forget to help the new guy.");

Assignee = assignee;

SendEmailToAssignee("You own it.");

}

public virtual void Deassigned()

{

if(Assignee != null) SendEmailToAssignee("You're off the hook.");

}

public virtual void SendEmailToAssignee(string message)

{

Console.WriteLine("{0}, RE {1}: {2}", Assignee, Title, message);

}

public virtual void Assign(string assignee){

\_rubyOperations.InvokeMember(Statemachine, "assign", assignee);

}

public virtual void Defer(){

\_rubyOperations.InvokeMember(Statemachine, "defer");

}

public virtual void Close(){

\_rubyOperations.InvokeMember(Statemachine, "close");

}

}

So we now have a completely compatible implementation of the Bug class or at least so we’d like to think, we still need to test it. I copied and pasted the code from listing 8.1 into the Main method of the console application we created in this chapter. I did change the names to make the outputs different.

+ivan@ivan-mbp:~/projects/ironruby-in-action/Samples/MonoBugTracker

(master)» mono MonoBugTrackerStatemachine.exe

Tom, RE The bug of bugs: You own it.

Tom, RE The bug of bugs: You're off the hook.

Ivan, RE The bug of bugs: You own it.

Ivan, RE The bug of bugs: You're off the hook.

Ivan, RE The bug of bugs: Don't forget to help the new guy.

Adam, RE The bug of bugs: You own it.

Adam, RE The bug of bugs: You're off the hook.

All done

Great! It looks like it all worked out nicely. So at this moment we’re compatible and our code will run on any .NET 2.0-enabled machine. At the moment of this writing the .NET framework just entered the Beta 2 stage for it’s 4.0 version. From a dynamic languages perspective this release is big. .NET 4.0 has a dynamic keyword which makes it possible to call methods on dynamic objects without the need to go through the object operations helper class. The next part of this chapter looks at what’s involved in making the bug sample work with .NET 4.0. If mono has progressed far enough by the time you read this they will also implement this dynamic keyword and so the next part should work on a mono box too, but for me it only works well on windows currently.

8.2.6 Leveraging the dynamic keyword in .NET 4.0

#TODO: finish this on windows.

8.3 Sweetening the plain vanilla FileSystemWatcher

The previous example was a good example of linking an existing Ruby library to a new or already existing CLR application. But we didn’t talk nearly as much about actually building a DSL as I’d like. I was presented with 2 options, either continue the example of the state machine or pick another subject to write a DSL around. I chose the latter because I get to show off much more stuff without the added complexity of logic surrounding state machines.

Something I will regularly do is: watch certain paths on the file system for changes and subsequently perform actions when something happens there. The .NET framework provides us with a FileSystemWatcher class to help you with that task. Unfortunately that FileSystemWatcher can be defined on a certain path, which can be a folder, and optionally you can set the watcher instance to include the subdirectories of that path too. But for any kind of more advanced filtering you’re either left with defining different paths to watch or provide your own filter when an event is fired. There are some other issues involved but in the interest of keeping the code simple; this version of the DSL won’t compensate for all of them, as that isn’t the topic of the discussion. Let’s start with determining where we want to take this.

8.3.1. Starting with the end result

We’ll soon dive in and get to leverage a bunch of meta-programming tricks to build the file watching DSL. In the DSL I’d like to use regular expressions for the filters as well as a limited form of glob patterns, limited in the sense that it only needs to understand the \*\*, ? and \* wildcard operators. I would like to specify multiple regular expressions for a given path. I would also like to be able to specify a “blanket” regular expression for a path that is used as the default filter and takes precedence over the more specific regular expressions. Listing 8.10 shows what the end result looks like for what I’d like to achieve.

Listing 8.10 The finished file system watch DSL

filesystem do

watch("/path/to/watch", /\_spec.rb$/ui) do

include\_subdirs

on\_change { |args| # do stuff here with args.path or args.name }

on\_change /integration\/.\*\_spec.rb/ { |args| # do stuff here }

on\_rename { |args| # do stuff here }

on\_rename "\*.rb" { |args| # do stuff here }

on\_delete { |args| # do stuff here with args.path or args.name }

on\_create { |args| # do stuff here with args.path or args.name }

on\_error { |args| # do stuff here with }

end

watch("/another/path/to/watch", /app\/.\*\.rb$/ui) do

top\_level\_only

on\_change { |args| # do stuff here with args.path or args.name }

on\_rename { |args| # do stuff here }

on\_delete { |args| # do stuff here with args.path or args.name }

on\_create { |args| # do stuff here with args.path or args.name }

on\_error { |args| # do stuff here }

end

end

As you can see we have an entry point called file system, which takes a block and then proceeds with defining new watch paths. Each watch path in turn takes another block where it defines the event handlers for this watch. This actual set up doesn’t really make sense in some cases but it does show the complete syntax we’re going for. For this example we won’t need any external libraries and so we can start right away with collecting the paths and filters.

8.3.2 Creating up the watcher builder

When I build this kind of library, I like to separate the configuration of the component from the actual creation of the component internally. I tend to use a common module defining the configuration syntax, which I share between a builder and the actual implementation. When the user starts using the DSL he’s actually talking to the builder and the builder object creates the implementation instance a little bit later. That way there is less chance of errors and the syntax is explicitly defined separately so that it’s easier for somebody else to learn it. And we’ll start with the parts of the builder component that can exist on their own without any other classes yet, listing 8.11 shows the output of the specs for that part.

Listing 8.11: Spec output for the first pass over the Builder

loading the library

- should raise an exception when IRONRUBY\_VERSION is undefined

WatcherSyntax

when initializing

- should raise an error when no path is given

- should have a path

- should have an empty filters collection when no filter is provided

- should register the filters

- should by default not recurse in subdirs

- should register and execute the block

when initialized

- should allow setting the path

- should allow setting extra filters

- should disable recursing into subdirs

- should enable recursing into subdirs

when registering handlers

- should register a handler without a filter

- should register a handler with a filter

- should register a handler by method name

The specs above are actually about the behavior that will be shared between the builder and the watcher implementation object. And it shows the intent of having filters at several levels, we need a path and it shows the behavior switch to recurse into sub directories.

It also describes the behavior of registering the handlers. So implementing this class would bring us pretty close to where we need to be in terms of syntax afterwards it’s a matter of providing an entry point and actually building the implementation.

Playing nice with other libraries

The library we are creating will only work on IronRuby. It might be good form to inform other ruby implementations that this library needs to be loaded from IronRuby to work.

There are several ways to solve that and most of them involve checking for a constant or the value of a constant. In this case I opted to check for the constant IRONRUBY\_VERSION, because I know that constant is unique for the IronRuby implementation.

Shri Borde of the IronRuby/DLR developer team recommends the following statement to check if you’re running in IronRuby:

do\_some\_ironruby\_stuff if defined?(RUBY\_PLATFORM) and RUBY\_PLATFORM == “ironruby”.

The latter way is probably the proper way of checking for the ruby platform you’re running on.

Because I know where this is heading I’m going to include this straight into a module. I started out with this as a part of the WatcherBuilder class and later extracted it into the WatcherSyntax module shown in Listing 8.12.

Listing 8.12: The common syntax module

module WatcherSyntax

attr\_accessor :path, :filters, :subdirs, :handlers

def path(val = nil) 1

@path = val if val

@path

end

def filter(\*val) 2

@filters = register\_filters @filters, \*val

end

def top\_level\_only 3

@subdirs = false

end

def recurse

@subdirs = true

end

alias\_method :include\_subdirs, :recurse

def on(action, \*filters, &handler) 4

@handlers ||= {}

hand = @handlers[action.to\_sym]

@handlers[action.to\_sym] = register\_handlers(hand, \*filters, &handler)

end

private

def register\_filters(coll, \*val)

val.inject(coll||[]) { |memo, filt| memo << filt unless memo.include?(filt); memo }

end

def register\_handlers(coll, \*filters, &handler)

hand = { :handlers => [], :filters => [] }.merge(hand||{}) 5

hand[:handlers] << handler

hand[:filters] = register\_filters hand[:filters], \*filters

hand

end

end

1. Overload the path attribute getter
2. Register filters
3. Switch for subdirectories
4. Register event handlers
5. Initialize default values in the data structure

The code in listing 8.12 first overloads the path getter (#1) that is created by the attr\_accessor method and when the provided value is not nil it will set the path value. Next we define the filter method (#2). This can be used to register extra filters. The actual registration of a filter is handled by a private method register\_filters that makes sure the new filters are only added if they don’t exist already. The next bit are actually 2 method where 1 is the inverse of the other to function as a switch to flick on recursing into subdirectories or not (#3). The last method is the on method, which is probably the most complex method in this chunk of code.

The reason that this code is more complex than it could be; is because I didn’t create a proper class to encapsulate a handler registration but am instead using a Hash as data structure. That has as a consequence that it also needs to be initialized with default values (#5). After initialization of the data structure it will add the handler to the handlers collection of the registration and it will register the filters in the filter collection for this handler.

If we were to include the module presented here at this point into a class we’d pass about 90% of the specs from listing 8.11. In the interest of keeping this chapter moving quickly, we’ll skip the intermediate implementation of the watcher builder and first move on to write some specs for the methods that are specific to a builder:

WatcherBuilder

when building watches

- should create a watcher

- should register a new watcher in the watcher bucket

- should register a new watcher and start it

This output of the specs mentions a watcher bucket. This is a class I did create for encapsulating watcher registrations. I chose to create a class in this case because we need extra behavior over the behavior of a standard Array, we want it to be able to stop and start the registered watchers and the map method should return a new WatcherBucket instead of an Array. For brevity this class and its specs aren’t included in the listings in this chapter but they are provided with the code samples for this book. Listing 8.13 shows the complete implementation of the builder class.

Listing 8.13: The completed builder

class WatcherBuilder

include WatcherSyntax 1

def initialize(path, \*filters, &configure) 2

@path = path

@filters = register\_filters [], \*filters

@subdirs = false

@handlers = {}

instance\_eval &configure if configure 3

end

def build

Watcher.new @path, @filters, @subdirs, @handlers 4

end

def method\_missing(name, \*args, &b) 5

if name.to\_s =~ /^on\_(.\*)/

self.on $1, \*args, &b

else

super

end

end

def self.watch(path, \*filters, &b)

@watchers ||= WatcherBucket.new 6

@watchers << WatcherBuilder.new(path, \*filters, &b).build

end

def self.build(&b) 7

@watchers = WatcherBucket.new

instance\_eval(&b)

@watchers.start\_watching

@watchers

end

end

1. Include the Syntax module
2. Initialize defaults
3. Call instance\_eval with extra config
4. Build an implementation object
5. Method missing for named handlers
6. Entry point for the DSL

The very first thing we do is include the WatcherSyntax module in the builder class (#1), this gives us then access to all the instance methods defined in that module. Next we setup a constructor method (#2) that will set up some defaults and the last thing it does is: call instance\_eval and pass it the &configure block (#3).

Using instance\_eval in this case is what gives us the clean syntax without defining a receiver for the methods. When you call instance\_eval in an object instance the contents of that block is evaluated with the receiver of self, that is self becomes the context of the block. We’ve seen this earlier in this chapter too, close to listing 8.4, when we were talking about the state machine. While this specific use makes sense if you want to go for a DSL, you should take care of using it because it makes your code less obvious.

After initialization of the builder we’ve defined a build instance method that creates a new instance of a Watcher implementation (#4). The last instance method that is defined is method\_missing (#5). You may have noticed that we’ve only defined the on method in the WatcherSyntax module, which takes an action as first argument. We’re going to leverage method\_missing as a method dispatcher for all methods that start with on\_. The remainder of the method name is assumed to be an action name and it gets dispatched to the on method. If it doesn’t start with on\_ then the call gets forwarded to the old behavior.

The last 2 methods are class methods, and both could serve as entry points into our DSL. The first one defines a single watch on a path with its handlers and adds those to the watcher bucket (#6) contained by the @watchers singleton variable. The last method build is the actual entry point for our DSL and can handle many calls to the watch thereby allowing the creation of multiple watches on multiple paths. All that is left to do now for us to get to the syntax shown in listing 8.10 is, define a global method that forwards its call to the build method of the WatcherBuilder class.

def filesystem(&b)

FsWatcher::WatcherBuilder.build(&b)

end

If we were to run the specs at this point they should all pass, allowing us to move on to the actual implementation of the Watcher class.

(master)» ibacon -q spec/\*\*/\*\_spec.rb

.....................

21 tests, 29 assertions, 0 failures, 0 errors

All specs pass, the natural order of the universe is preserved, in short all is well with the world. We now have a configuration DSL that can build watchers and start them if only they would have been implemented. The next and last part of this chapter talks about implementing the actual functionality.

8.3.3 Implementing the actual watcher

At this point we’re able to configure a watcher but it won’t do anything useful because the implementation of that class is still open on the to-do list.