[9.1 Easier GUI building for Windows Forms 2](#_Toc243032709)

[9.1.1 Trying the simplest thing first 2](#_Toc243032710)

[9.1.2: Adding a menu without a DSL 4](#_Toc243032711)

[9.1.3 Removing the clutter with a menu DSL 4](#_Toc243032712)

[9.1.4 Cleaning up control instance creation 7](#_Toc243032713)

[9.1.5 Moving towards a DSL for Forms 8](#_Toc243032714)

[9.1.6 Preparing for packaging 11](#_Toc243032715)

[9.1.8 Final thoughts 12](#_Toc243032716)

[9.2 Extending AutoCAD with IronRuby 12](#_Toc243032717)

[9.2.1 AutoCAD and dynamic programming 13](#_Toc243032718)

[9.2.2 Getting the base infrastructure down 16](#_Toc243032719)

[9.2.3 Overruling as we go 18](#_Toc243032720)

[9.2.4 Transforming overrules 21](#_Toc243032721)

[9.2.5 Commands for the UI 23](#_Toc243032722)

[9.2.6 Executing the code 26](#_Toc243032723)

[9.3 Summary 30](#_Toc243032724)

9

Tales from the community

This chapter covers:

* Building a WinForms DSL
* Creating a gem
* Extending AutoCAD
* A case study on some advanced usage

When I started this book my publisher told me: this book needs to be useful for the readers. Some of the most useful things for me when I’m learning something new is to get different points of view on a particular subject, and to talk to people that are experienced in the subject. That is what this chapter is all about, I’ve gathered a few contributions from the IronRuby community for exactly this reason.

In this chapter, my colleague Thibaut Barrère takes us through how he builds a DSL for creating winforms GUIs. When we’ve learned how to do, that we’ll meet colleague Kean Walmsley as he explains about the ins and outs of extending AutoCAD with IronRuby.

This chapter is completely IronRuby specific because none of the described applications of the Ruby language can be done with MRI for example. Winforms is a technology that is specific to .NET as is the extensibility in AutoCAD.

The first person to speak, Thibaut, is describing how IronRuby helps him to quickly turn around prototypes he builds for his clients. From here on, I’ll let Thibaut do the talking, and we’ll meet again to summarize his experience.

9.1 Easier GUI building for Windows Forms

I’ve been playing around with IronRuby to make it easier to write Windows Forms code. As I experimented, I gradually extracted a gem called Magic – a DSL targeting the creation of .NET-based GUIs. Magic's main focus is Windows Forms. You can find the magic DSL in my github account (<http://github.com/thbar/magic>)

This section describes the various steps that led to Magic, rather than just presenting the DSL, so that newcomers to Ruby have a better chance to learn something useful. There is a saying that goes: Give a man a fish and you feed him for a day, teach that man how to fish and feed him for a lifetime. I’d much prefer the lifetime option.

I chose to explain this process in this manner because, in my experience, working incrementally by trying out things, factoring code out as I go and keeping what makes sense is especially easy and common when working in Ruby as compared to C#.

From a C# perspective, VB.NET may be an anti-pattern. From an IronRuby perspective, both of those languages may be dinosaurs.

Scott Bellware - <http://twitter.com/bellware/status/804236487>

The first thing I did when I started my voyage was to create a simple form with a button, without any DSL trickery.

9.1.1 Trying the simplest thing first

The first thing I wanted to find out was if I could actually create windows forms applications with IronRuby. So I decided to build a simple form that has a button that displays a message box. That would tell me how to set properties on Windows Forms objects and how to hook up an event. I realized that during my experiments I would require and include some common logic so I started by creating a common.rb file, listing 9.1, to group that logic together.

Listing 9.1: Grouping common logic together

require 'mscorlib' 1

require "System, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

require 'System.Windows.Forms, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'

require 'Microsoft.Scripting, Version=1.0.0.2000, Culture=neutral, PublicKeyToken=null'

require 'IronRuby, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null'

include System 2

include System::Windows::Forms

include System::Collections

include System::ComponentModel

1. require assemblies

2. add shortcuts for namespaces

The first bit of code first loads the assemblies we’re going to need when working with forms, much like when you would add references for a visual studio project #1. The other bit of code contains mostly include statements. They can be considered using statements, including those provides access to the types defined in those namespaces,without the need to prefix them with the complete namespace #2.

Now that I’ve gotten my common logic in place, I created a new file, shown in listing 9.2, form\_and\_button.rb. This file contains the code needed to create the simple Windows forms application.

Listing 9.2: The first attempt at a windows forms application

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

form = Form.new 1

form.text = "Hello world!"

container = FlowLayoutPanel.new 2

container.dock = DockStyle.fill

form.controls.add(container)

button = Button.new 3

button.text = "Click!"

button.click do |sender,args| 4

MessageBox.show("Oh, I'm clicked...")

end

container.controls.add button

Application.run(form) 5

1. Create form instnce

2. Create layout panel

3. Create button

4. Attach event to the button

5. Run application

In this code listing I first create a new Form instance #1, and I set its text to “Hello world!”. Next I add a FlowLayoutPanel instance as a container #2 so I can add more elements to the form. Next, I create a button #3 and attach an event handler #4 to the instance that will show a message box with “Oh, I’m clicked…” We add the button to the container and run the application with the form. #5

To run this application, open a console, navigate to where you created the ruby files and execute the command: ir form\_and\_button.rb. Figure 9.1 shows the end result of this script.

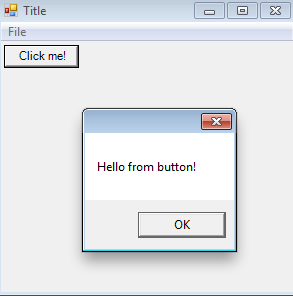


Figure 9.1: Hello World for the magic DSL

So far, we’ve got a working form, but it isn’t inconceivable that we might want to add a menu at some point so I tried adding that without any form of DSL as well.

9.1.2: Adding a menu without a DSL

At this point, I just want to find out what it would be like to create a simple menu with some event handlers defined for the click event. To do that, I created a file called form\_and\_menu.rb, as shown in listing 9.3.

Listing 9.3: Exploring building a menu

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

form = Form.new

menu = MainMenu.new

fileItem = menu.menu\_items.add("&File")

newItem = fileItem.menu\_items.add("&New")

newItem.menu\_items.add("Spreadsheet")

newItem.menu\_items.add("Document")

fileItem.menu\_items.add("&Quit").click { |s,e| Application.Exit }

toolsItem = menu.menu\_items.add("&Tools")

toolsItem.menu\_items.add("&PowerBlade").click { |s,e| MessageBox.Show("Powerblades are amazing...") }

toolsItem.menu\_items.add("&Scissors")

form.menu = menu

Application.Run(form)

This code works and it does create a menu, but it left me a bit unsatisfied. The first thing that bothers me is that this code is full of clutter from things like menu\_items.add and variable references. I needed to indent the code to maintain an overview of the level at which I was adding menu items. Having to resort to indentation and previously created instances is error-prone and uncomfortable, especially when the codebase grows larger and more complex.

It’s not all bad though; when adding a menu item with menu\_items.add it returns the newly created menu item instance, which is handy for chaining a click handler onto it.

I’m mostly bothered by the need for temporary variables and the clutter this introduces. Perhaps ruby blocks could be a solution. And this is exactly what I’ll be trying next.

9.1.3 Removing the clutter with a menu DSL

To get started and figure out the direction I want to end up with, I started by writing out the API I’d like to achieve and then later on implement the code necessary to get to that API.

In listing 9.4, I’ve written out the code as I’d like it to read. It creates the same menu from Listing 9.3, but it should read easier. There will be a few new concepts introduced in listing 9.4, which I’ll explain.

Listing 9.4: The desired menu DSL example usage

$LOAD\_PATH << File.dirname(\_\_FILE\_\_) 1

require 'common'

require 'menu\_builder'

form = Form.new

form.menu = MainMenu.build do 2

item("&File") {

item("&New") {

item("Spreadsheet")

item("Document")

}

item("&Quit").click { Application.Exit }

}

item("&Tools") {

item("&PowerBlade").click { MessageBox.Show("Powerblades are amazing...") }

# alternate form, with lambda instead of chained call

item "&Scissors", lambda { MessageBox.Show("Scissors are nice, too") }

}

end

Application.Run(form)

1. Add current dir to the load path
2. Entrypoint in the DSL

The first thing I do is add the current directory to the search paths of IronRuby so I don’t have to specify full file paths for the requires of my files #1. I then require the common file and the menu\_builder file, shown in listing 9.5. Then I create a new instance of a form and assign the result of the DSL to the menu property of the form instance #2.

In this DSL, blocks delimit the hierarchies in the menu structure now. It’s also clearer for me to add items without having to deal with temporary instance variables, making it less error-prone.

Listing 9.5: A first attempt at a DSL

# a first attempt at dsl-ing windows forms menu creation from IronRuby

# inspiration from http://blog.jayfields.com/2006/07/ruby-block-scope.html

class MainMenu

class << self

def build(&script) 1

instance = self.new

instance.instance\_eval(&script) 2

instance

end

end

# add a new menu item.

# to receive click event:

# - pass an optional lambda { }

# - or chain a .click { } call after item()

def item(label,click\_handler=nil)

@parents ||= [self] 3

@parents.push(@parents.last.menu\_items.add(label))

@parents.last.click { click\_handler.call } if click\_handler 4

yield if block\_given? 5

@parents.pop

end

end

1. Factory method
2. Instance\_eval call
3. Initialize stack
4. Attach click handler
5. Call the provided block

Although it may all look cryptic and abstract at first, it’s a lot easier than it looks. I wrote this code and I am by no means a meta-programming expert. A number of blog posts illustrate the instance\_eval trick I’ve opted to use here.

We create an instance of the MainMenu class with the factory method build #1. This method takes a block parameter and evaluates it in the context of MainMenu. This means that all the calls to item inside the block will be sent to the MainMenu instance #2.

In the item method I use an Array as a stack to store the added menu items and to keep track of the hierarchy #3. This array is stored as an instance variable. The item method also takes an optional click\_handler parameter, when this is passed in it is registered to the click event, by wrapping it in another block #4. There is no & prefix for the click\_handler parameter because we will still allow to pass a block in. When this block is passed it gets evaluated to give children the chance to register themselves #5. The corresponding parent will always be on top of the stack.

Be careful with instance\_eval

While instance\_eval is a very cool tool to have you should use it with caution. It moves the code you are executing to the instance that receives the block. It is generally a more advisable approach to use a block and then pass the instance as a parameter to the block. This does hurt readability a little, but it will make it a lot clearer when you’re revisiting the code in a couple of months.

This doesn’t mean you shouldn’t use instance\_eval, just that you should understand why you’re using it. That being said, instance\_eval may be appropriate here.

I would personally vote for including it in a little note with the rdoc comment above the method when you do use it. That way people can still be warned if they read the documentation or their IDE displays the tooltip.

- Ivan

This is definitely a step in the right direction; now I have a pretty good and reusable way of creating menus with code that is a lot clearer than before. However there are still a bunch of simplifications and improvements possible. But it is good enough for now so I start thinking about the following problem I want to tackle. Is there some way to clarify the code needed to set properties on a control on instantiation? That is what I’ll be exploring next, decluttering control instantiation.

9.1.4 Cleaning up control instance creation

In the ruby world, it’s quite common to pass a hash to a constructor, and this hash is used to initialize the properties of that instance. I wanted some of that in my API too. Again, I started off by creating the code, shown in listing 9.6, that I wanted my API to look like at this stage. I created a new file called form\_sugar.rb.

Listing 9.6: The control builder API

$LOAD\_PATH << File.dirname(\_\_FILE\_\_)

require 'common'

require 'control'

form = Form.build(:text => "Hello!") 1

container = FlowLayoutPanel.build(:dock => DockStyle.fill)

container << Button.build(:text => "Click me!", :click => lambda { MessageBox.Show("Hello") })

container << Button.build(:text => "Quit", :click => lambda { Application.Exit })

form << container 2

Application.Run(form)

1. Use of the build method
2. Use << instead of controls.add

This time I don’t create a form instance by calling the new method, but instead I use the method build #1. I then add a FlowLayoutPanel in the same way. The container variable with a FlowLayoutPanel instance then replaces controls.add with << #2. This << operator is used to add 2 buttons to the panel. Both buttons add a different click handler. And lastly the container is added to the form and the application runs the form.

This is one of those moments when you’ll completely love ruby. This type of thing seems very complicated when you come from a language like C#, but in a language like IronRuby this turns out to be a piece of cake as shown in listing 9.7.

Listing 9.7: Some syntactic sugar for building controls

class Control 1

# allows Button.build(:text => "Yeah")

def self.build(options={})

control = self.new

options.each do |k,v|

if v.is\_a?(Proc) 2

control.send(k) { v.call } 3

else

control.send("#{k}=",v) 4

end

end

control

end

def <<(control) 5

controls.add(control)

self

end

end

1. Monkey-patch System::Windows::Control
2. Check if proc
3. Attach as event handler
4. Use as setter
5. Define << method

All I needed to do to enable the build factory method for any control that subclasses System.Windows.Control was monkey-patch the class Control #1. This factory method first creates a new instance of the control by calling its new method. Next it iterates over the pairs in the hash. I check if the value is a Proc #2 because then we need to pass the block to the method in order to hook up the event handler #3, otherwise we send the value to a setter #4. This approach takes advantage of the fact that Ruby is based on message passing. It can also pass messages by sending them to object instances. There is also an instance method defined in this monkey-patch. This method defines the << operator and adds the parameter to the controls collection of this control instance #5.

I’m glad to see that I could remove some more of the clutter. It can be duplicated and transformed quite easily. So far, I can create menu items with a DSL and I have a builder for objects that takes a hash. The next step is to move to a DSL for windows forms by transposing the menu items DSL on the control creation.

9.1.5 Moving toward a DSL for Forms

To create a DSL for building a windows form control, I have to find a way to specify many different control classes to create an instance from. I’ve got a multitude of solutions available for me to aid in this situation. The simplest option is to go with a method\_missing hook. Using method\_missing comes at a performance cost so it should be used with caution.

As I did with the previous experiments I first write down where I want to get to, as shown in listing 9.8, and later I’ll implement the code to get to that API.

Listing 9.8: The control building DSL

$LOAD\_PATH << File.dirname(\_\_FILE\_\_)

require 'common'

require 'magic'

require 'menu\_builder'

form = Magic.build do

form(:text => "Hello") do

flow\_layout\_panel(:dock => DockStyle.fill) do

button(:text => "Click me!", :click => lambda { MessageBox.Show("Hello") })

button(:text => "Quit", :click => lambda { Application.Exit })

end

end

end

form.menu = MainMenu.build do

item("&File") {

item("&Quit").click { Application.Exit }

}

end

Application.run(form)

I want to get to the point where the block that gets passed to the Magic.build factory method will result in a GUI with that hierarchy. To get there, I’ll extract the control building into its own module. The code is very similar to the one found in listing 9.7. I’ll also look into using the method\_missing as shown in listing 9.9.

Listing 9.9: The first draft of the Magic DSL

module InstanceCreator 1

# instantiate the given class and set the properties passed as options

# to support both values and procs for options

def build\_instance\_with\_properties(klass,properties={})

instance = klass.new

properties.each do |k,v|

if v.is\_a?(Proc)

instance.send(k) { v.call }

else

instance.send("#{k}=",v)

end

end

instance

end

end

# a better attempt at Windows::Forms dsl

class Magic

include InstanceCreator 2

class << self

def build(&description)

self.new.instance\_eval(&description)

end

end

def classify(string)

string.gsub(/(^|\_)(.)/) { $2.upcase }

end

def method\_missing(method,\*args)

@stack ||= [] 3

clazz = Object.const\_get(classify(method.to\_s)) 4

control = build\_instance\_with\_properties(clazz, \*args)

# add to the parent control - only if it is a control

# useful to declare background\_worker with the same syntax

@stack.last.controls.add(control) if (@stack.last && control.is\_a?(Control)) 5

@stack.push(control)

yield if block\_given?

@stack.pop

end

end

1. Create InstanceCreator module
2. Include in Magic class
3. Initialize stack
4. Create class from name
5. Add to parent if appropriate

I’ve extracted the control building in an InstanceCreator module #1. Then I created a new class called Magic. This class includes the InstanceCreator module #2. It has a build class method that creates a new instance and evaluates the block in the context of that instance.

The most interesting method in the Magic class is definitely the method\_missing implementation. This method receives the method name and its arguments. I’ll take you through line by line.

The first thing I do is initialize the stack if it doesn’t exist already #3. Then I need to convert the snake cased (flow\_layout\_panel) string to a camel cased string (FlowLayoutPanel). To do that I pass the method\_name to the classify method which will convert the string into it’s camel cased counterpart. Now this string needs to be converted to a class constant for Ruby. I do that by calling Object.const\_get with the result of the classify method. At this point I’ve got the control class I want to build an instance from #4.

I pass the control class along with any arguments (like the property hash) to the method build\_instance\_with\_properties from the InstanceCreator module to create an instance an initialize the properties of that instance.

Now that I have an instance of a class it’s time to add it to the necessary collections. When the class is a subclass of System.Windows.Control I need to add it to the controls collection of the parent object #5. All instances are added to the stack too. When a block is given it will be evaluated at this point. The last thing I do is pop the last control that got put on the stack so that I’m always talking to the correct control.

At this point I’ve got all the building blocks I need to create my windows forms DSL. Unfortunately I still have 2 different ways to start building components. I now have Magic and MainMenu as entry points into my DSL. This next part explains how to get around that.

9.1.6 Preparing for packaging

Before Magic is complete enough to put it in its own project, I’d like to unify the way to access the DSL implementations. The differences between the 2 implementations are that to add menu items I need to get to the menu\_items property of the parent and for controls I need to get to the controls property of the parent. Let’s resort to some duck typing techniques for that. Listing 9.10 shows the new Magic class with the changes marked in bold.

Listing 9.10: The final magic implementation

# a better attempt at Windows::Forms dsl

class Magic

include InstanceCreator

class << self

def build(&description)

self.new.instance\_eval(&description)

end

end

def classify(string)

string.gsub(/(^|\_)(.)/) { $2.upcase } # simplified version of Rails inflector

end

def method\_missing(method,\*args)

@stack ||= []

parent = @stack.last

clazz = Object.const\_get(classify(method.to\_s))

**instance** = build\_instance\_with\_properties(clazz, \*args)

# Windows Forms Control and MenuItem support

if defined?(System::Windows::Forms::Control) or defined?(System::Windows::Forms::MenuItem)

parent.controls.add(instance) if **parent.respond\_to?(:controls) && instance.respond\_to?(:create\_control)**

parent.menu\_items.add(instance) if **parent.respond\_to?(:menu\_items)**

end

@stack.push(instance)

yield if block\_given?

@stack.pop

end

end

For the duck-typing to work I cheated and looked at which methods I need and if the parent has a controls method then I’m assuming we’re dealing with a control. If the parent responds to a menu\_items method we can add the instance there. The enclosing if statement ensures that we’re using windows forms controls or menu items.

And that concludes our DSL for Windows Forms. Before I summarize what has been discussed here I’d like to leave you with an example of the usage of this DSL. This is shown in listing 9.11.

Listing 9.11: The end result of the Magic DSL

form = Magic.build do

form(:text => "DataGridView sample", :width => 800, :height => 600) do

grid = data\_grid\_view :dock => :fill

grid.column\_count = 2

grid.columns[0].name = "First name"

grid.columns[1].name = "Last name"

grid.rows.add("John","Smith")

grid.rows.add("Barbara","Carpenter")

end

end

I didn’t include the support for enums in the discussion above nor did I include any discussion about unit tests. I’ve had to drop these to reach the space requirements for this discussion. You can find the code for the magic gem in my [github account](http://github.com/thbar/magic).

9.1.7 Final thoughts

As someone who has developed and maintained a reasonably sized Windows Forms applications (~500K LOC), I have the gut feeling that IronRuby has quite a lot to offer. I was pleasantly surprised to see how things evolved while working on this DSL. The best thing was that although it seemed very hard at the beginning it was actually a lot of fun.

As long as you keep in mind that you need a good bit of refactoring and a high-level of test coverage to keep things under control, it is my belief that it is quite easy to factor things out and create a modular application with the following features:

* Simple hooks for i18n
* Common builder code to share properties across the forms (fonts, etc)
* Plug-ins and generic forms
* State saving

That’s all for me, I hope you enjoyed my contribution.

Thibaut Barrère – <http://logeek.fr> / <http://github.com/thbar>

9.2 Extending AutoCAD with IronRuby

The previous section had been contributed by Thibaut Barrère and dealt with the Windows Forms technology and how that can make your coding life more pleasant. Kean Walmsley, a manager over at Autodesk, contributes the second half of this chapter. He will discuss what is involved to get IronRuby and AutoCAD to become friends. This contribution uses the DLR hosting facilities to extend AutoCAD with ruby scripting capabilities.

To follow along with the samples that will be presented in this section, you’ll need to have AutoCAD installed on your system. I downloaded a trial to run these samples when I was helping Kean out to find some workarounds for some things that hadn’t yet been implemented when he was working on this. From here on out it will be Kean talking.

9.2.1 AutoCAD and dynamic programming

AutoCAD has a long history of supporting dynamic programming: since the very early days of the software – and the early days means the mid-80s, something to which few mainstream applications available today can lay claim – AutoCAD has provided a fully-integrated LISP interface, allowing people to customize and automate the product by entering dynamic code at AutoCAD’s command-line, it’s built-in REPL. The availability of AutoLISP was key to driving AutoCAD’s early success and it remains one of the most widely-used LISP implementations on the planet.

While the amount of LISP code in production use today numbers in the millions of lines, much of AutoCAD’s development community has moved on through the worlds of C++ (for professional developers) and VBA (for everyone) to the incredibly compelling experience of programming with the .NET Framework. AutoCAD has included a managed API since its 2005 release, and this is now considered by many to be the platform of choice when it comes to programming with AutoCAD.

AutoCAD’s .NET API is extremely rich and provides compelling capabilities for extending and automating the product, whether programming in VB.NET, C# or newer members of the .NET family such as F#, IronPython and IronRuby. This section focuses on how best to take advantage of IronRuby from within AutoCAD. We’ll start by exposing a simple command to allow loading and execution of Ruby scripts within AutoCAD and then move on to using some of the latest AutoCAD API functionality from IronRuby.

We’re going to do some cool stuff with AutoCAD in a moment but before we do that I’d like to show you where we start (figure 9.2) and where we will end up (figure 9.3).

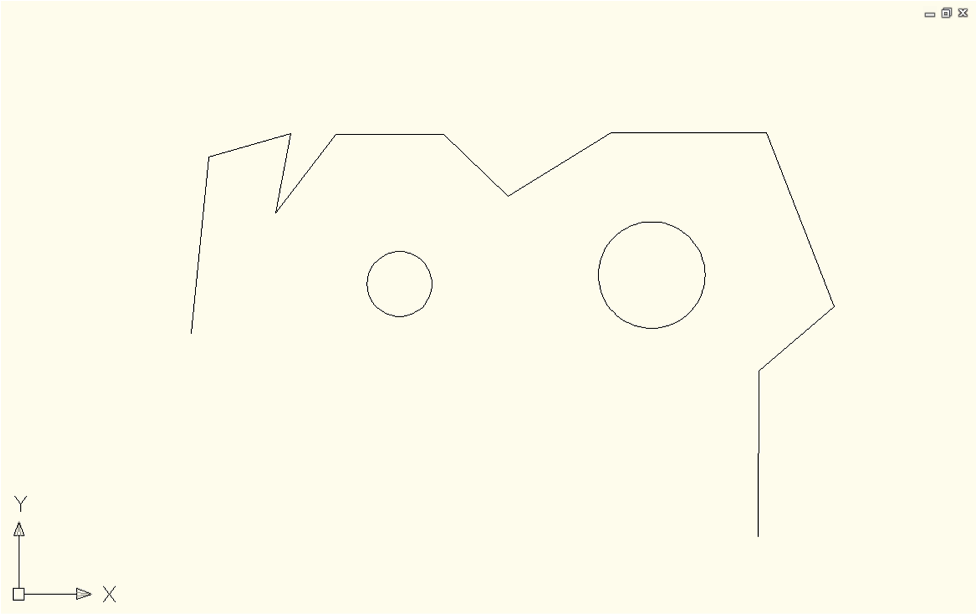


Figure 9.2: the starting point

The end result is that the lines are turned into pipes.

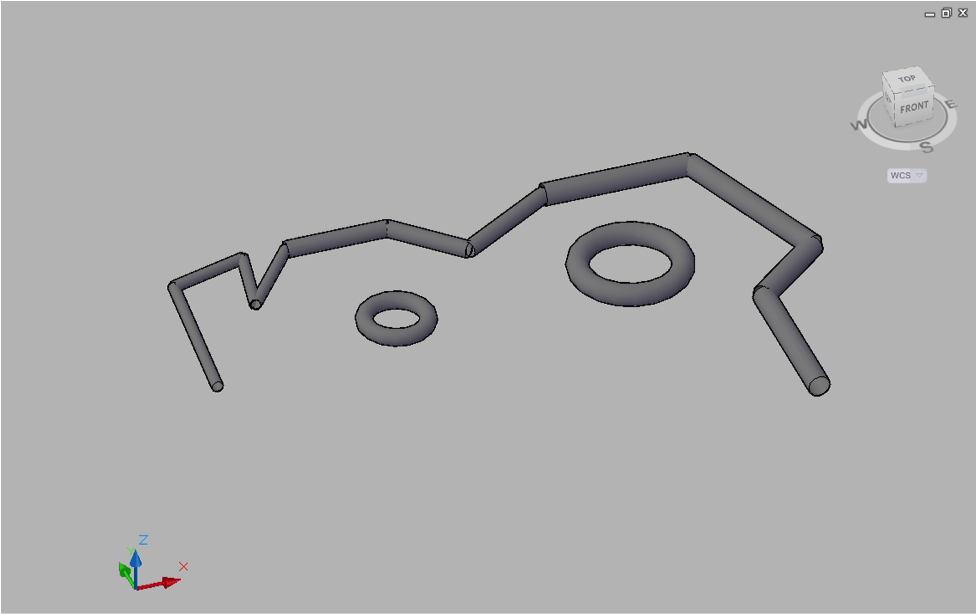


Figure 9.3: the end result

Introducing AutoCAD to IronRuby

In order to enable use of AutoCAD’s .NET interface from IronRuby, we’re going to need to host the IronRuby engine and use it to execute scripts. Traditional AutoCAD .NET applications are compiled DLLs – loadable into AutoCAD using its NETLOAD command – but as we’re dealing with interpreted scripts we will need to implement a parallel mechanism, an RBLOAD command. As we’re also going to want to register our own commands from our scripts, we’ll be making use of an undocumented technique, shown in listing 9.12, for dynamic command registration (the traditional approach is to mark your command methods using custom .NET attributes, something that is not available in IronRuby).

Listing 9.12: Teaching AutoCAD to speak IronRuby

using Autodesk.AutoCAD.ApplicationServices;

using Autodesk.AutoCAD.DatabaseServices;

using Autodesk.AutoCAD.Runtime;

using Autodesk.AutoCAD.EditorInput;

using IronRuby.Hosting;

using IronRuby;

using Microsoft.Scripting.Hosting;

using System.Reflection;

using System;

namespace RubyLoader

{

public class Commands

{

[CommandMethod("-RBLOAD")]

public static void RubyLoadCmdLine() 1

{

RubyLoad(true);

}

[CommandMethod("RBLOAD")] 2

public static void RubyLoadUI()

{

RubyLoad(false);

}

public static void RubyLoad(bool useCmdLine)

{

Document doc = Application.DocumentManager.MdiActiveDocument;

Editor ed = doc.Editor;

short fd = (short)Application.GetSystemVariable("FILEDIA");

// As the user to select a .rb file

PromptOpenFileOptions pfo = new PromptOpenFileOptions(

"Select Ruby script to load");

pfo.Filter = "Ruby script (\*.rb)|\*.rb";

pfo.PreferCommandLine = (useCmdLine || fd == 0);

PromptFileNameResult pr = ed.GetFileNameForOpen(pfo);

// And then try to load and execute it

if (pr.Status == PromptStatus.OK)

ExecuteRubyScript(pr.StringResult);

}

[LispFunction("RBLOAD")] 3

public ResultBuffer RubyLoadLISP(ResultBuffer rb)

{

const int RTSTR = 5005;

Document doc = Application.DocumentManager.MdiActiveDocument;

Editor ed = doc.Editor;

if (rb == null)

{

ed.WriteMessage("\nError: too few arguments\n");

}

else

{

// We're only really interested in the first argument

Array args = rb.AsArray();

TypedValue tv = (TypedValue)args.GetValue(0);

// Which should be the filename of our script

if (tv != null && tv.TypeCode == RTSTR)

{

// If we manage to execute it, let's return the

// filename as the result of the function

// (just as (arxload) does)

bool success =

ExecuteRubyScript(Convert.ToString(tv.Value));

return

(success

? new ResultBuffer(new TypedValue(RTSTR, tv.Value))

: null);

}

}

return null;

}

private static bool ExecuteRubyScript(string file) 4

{

// If the file exists, let's load and execute it

bool ret = System.IO.File.Exists(file);

if (ret)

{

try

{

LanguageSetup ls = Ruby.CreateRubySetup();

ScriptRuntimeSetup rs = new ScriptRuntimeSetup();

rs.LanguageSetups.Add(ls);

rs.DebugMode = true;

ScriptRuntime runtime = Ruby.CreateRuntime(rs);

runtime.LoadAssembly(

Assembly.GetAssembly(typeof(Commands))

);

ScriptEngine engine = Ruby.GetEngine(runtime);

engine.ExecuteFile(file);

}

catch (System.Exception ex)

{

Document doc =

Application.DocumentManager.MdiActiveDocument;

Editor ed = doc.Editor;

ed.WriteMessage(

"\nProblem executing script: {0}", ex

);

}

}

return ret;

}

}

}

Once built into a Class Library and loaded into AutoCAD using NETLOAD, this code will expose two commands and one function:

* RBLOAD a command that allows the user to select a script using a file selection dialog #2
* -RBLOAD the same as the RBLOAD command, but uses command-line input #1
* RBLOAD the equivalent AutoLISP-callable function #3

All these entry-points call into the same shared function, ExecuteRubyScript(), which hosts the IronRuby scripting environment inside AutoCAD, and ask it to execute the specified “.rb” file #4. From here on, we can venture into Ruby. We’re going to use the new Overrule API in AutoCAD 2010, and to do that we need a little bit of infrastructure.

9.2.2 Getting the base infrastructure down

Now on to our Ruby script, from which we want to make use of the new Overrule API in AutoCAD 2010. This API allows us to customize the behavior of any AutoCAD drawing object, and in this simple example, we’ll use it to customize the display of lines and circles. The lines and circles (figure 9.2) would be given a radius, effectively turning them into pipes (figure 9.3). This radius is going to be stored with the object as Extended Entity Data (XData), a common technique in AutoCAD for persisting data with the built-in objects provided with the product. Listing 9.13 shows how we’re referencing the .NET assemblies and defining shortcuts to the namespaces we’ll be using from AutoCAD.

Listing 9.13: Referencing libraries and namespaces

require 'acmgd.dll'

require 'acdbmgd.dll'

Ai = Autodesk::AutoCAD::Internal

Aiu = Autodesk::AutoCAD::Internal::Utils

Aas = Autodesk::AutoCAD::ApplicationServices

Ads = Autodesk::AutoCAD::DatabaseServices

Aei = Autodesk::AutoCAD::EditorInput

Agi = Autodesk::AutoCAD::GraphicsInterface

Ag = Autodesk::AutoCAD::Geometry

Ac = Autodesk::AutoCAD::Colors

Ar = Autodesk::AutoCAD::Runtime

These namespace aliases give us easy access to the common namespaces we’re going to use in our script. The next step is to define a function that calls through to AutoCAD’s editor to display messages in the command-line, as shown in listing 9.14.

Listing 9.14: Printing messages inside AutoCAD

def print\_message(msg)

app = Aas::Application

doc = app.document\_manager.mdi\_active\_document

ed = doc.editor

ed.write\_message(msg)

end

When we load our script, we’re going to want to register a number of commands allowing AutoCAD users to call our functions. The standard approach for doing this is via a custom .NET attribute, but as that capability is not available to us we’re going to call an “internal” function to register a command shown in listing 9.15 (if using versions of AutoCAD after 2010, you may find this function in an officially support namespace).

Listing 9.15: Registering commands in AutoCAD

# Function to register AutoCAD commands

def autocad\_command(cmd)

cc = Ai::CommandCallback.new method(cmd)

Aiu.add\_command('rbcmds', cmd, cmd, Ar::CommandFlags.Modal, cc)

# Let's now write a message to the command-line

print\_message("\nRegistered Ruby command: " + cmd)

end

def add\_commands(names)

names.each { |n| autocad\_command n }

end

Once we have defined our command functions, we’ll call add\_commands() passing a list of the function names for these commands to be registered in our AutoCAD session.

Listing 9.16 show the code we need to define some functions to provide access to our XData – both to read and to write it.

Listing 9.16 Reading and writing XData

APP\_NAME = "TTIF\_PIPE"

APP\_CODE = 1001

RAD\_CODE = 1040

def pipe\_radius\_for\_object(obj)

res = 0.0

begin

rb = obj.x\_data

return res if rb.nil?

rb.each do |tv|

foundStart = tv.type\_code == APP\_CODE and tv.value == APP\_NAME

if foundStart and tv.type\_code == RAD\_CODE

res = tv.value

break

end

end

rescue

return 0.0

end

return res

end

def set\_pipe\_radius\_for\_object(tr, obj, radius)

# Set the pipe radius as XData on a particular object

db = obj.database

# Make sure the application is registered

# (we could separate this out to be called

# only once for a set of operations)

rat = tr.get\_object(db.reg\_app\_table\_id, Ads::OpenMode.for\_read)

unless rat.has(APP\_NAME)

rat.upgrade\_open()

ratr = Ads::RegAppTableRecord.new

ratr.name = APP\_NAME

rat.add(ratr)

tr.add\_newly\_created\_d\_b\_object(ratr, true)

end

# Create the XData and set it on the object

rb = Ads::ResultBuffer.new(

Ads::TypedValue.new(APP\_CODE, APP\_NAME),

Ads::TypedValue.new(RAD\_CODE, radius))

obj.x\_data = rb

rb.dispose

end

In the code above we’ve got a method called: pipe\_radius\_for\_object, and is responsible for getting the pipe radius from the XData and returning that if any. The set\_pipe\_radius\_for\_object method creates an XData object and uses that to set the pipe radius. This is the entire infrastructure we’re going to need to start implementing the Overrule classes.

9.2.3 Overruling as we go

Now we can start implementing our overrules. We’ll start with a DrawableOverrule, which unsurprisingly overrules the behavior of drawables (an AutoCAD base class for objects such as lines and circles). We’re going to implement our own, very simple base class for our circle and line drawable overrules, as they will both call the SetXDataFilter() method when they initialize to tell AutoCAD the name of the application registering the XData (shown in listing 9.17). We’re passing “TTIF\_PIPE” to this function (TTIF stands for Through the Interface, the blog I write on AutoCAD development – <http://blogs.autodesk.com/through-the-interface>).

Listing 9.17: The DrawableOverrule base implementation

class PipeDrawOverrule < ConcreteClasses::DrawableOverrule

def initialize

set\_x\_data\_filter($appName)

end

end

We’ll define two classes, listing 9.18, based on PipeDrawOverrule, one for lines and one for circles. These classes will implement world\_draw() functions which will get called whenever AutoCAD wants to generate graphics for our objects. This is our opportunity to overrule the standard behavior and draw solid objects – whether extruded along a linear path or swept along a circular one.

Listing 9.18: The overrules

class LinePipeDrawOverrule < PipeDrawOverrule

# An overrule to make a pipe out of a line

def initialize

@sweepOpts = Ads::SweepOptions.new

super

end

def world\_draw(d, wd)

radius = pipeRadiusForObject(d)

if radius > 0.0

# Draw the line as is, with overruled attributes

parent\_world\_draw(d, wd)

unless d.id.is\_null and d.length > 0.0

# Draw a pipe around the line

c = wd.sub\_entity\_traits.true\_color

wd.sub\_entity\_traits.true\_color =

Ac::EntityColor.new 0x00AFAFFF

wd.sub\_entity\_traits.line\_weight =

Ads::LineWeight.line\_weight\_000

start = d.start\_point

endpt = d.end\_point

norm = Ag::Vector3d.new(

endpt.X - start.X,

endpt.Y - start.Y,

endpt.Z - start.Z)

clr = Ads::Circle.new start, norm, radius

pipe = Ads::ExtrudedSurface.new

begin

pipe.create\_extruded\_surface(clr, norm, @sweep\_opts)

rescue

print\_message "\nFailed with CreateExtrudedSurface."

end

clr.dispose()

pipe.world\_draw(wd)

pipe.dispose()

wd.sub\_entity\_traits.true\_color = c

end

return true

end

return super

end

def set\_attributes(d, t)

i = parent\_set\_attributes(d, t)

radius = pipe\_radius\_for\_object(d)

if radius > 0.0

# Set color to magenta

t.color = 6

# and lineweight to .40 mm

t.line\_weight = Ads::LineWeight.line\_weight\_040

end

return i

end

end

class CirclePipeDrawOverrule < PipeDrawOverrule

# An overrule to make a pipe out of a circle

def initialize

@sweep\_opts = Ads::SweepOptions.new

super

end

def world\_draw(d, wd)

radius = pipe\_radius\_for\_object(d)

if radius > 0.0

# Draw the circle as is, with overruled attributes

parent\_world\_draw(d, wd)

# Needed to avoid ill-formed swept surface

if d.radius > radius

# Draw a pipe around the circle

c = wd.sub\_entity\_traits.true\_color

wd.sub\_entity\_traits.true\_color =

Ac::EntityColor.new 0x3FFFE0E0

wd.sub\_entity\_traits.line\_weight =

Ads::LineWeight.LineWeight000

start = d.StartPoint

cen = d.Center

norm = Ag::Vector3d.new(

cen.X - start.X,

cen.Y - start.Y,

cen.Z - start.Z)

clr =

Ads::Circle.new start, norm.cross\_product(d.normal), radius

pipe = Ads::SweptSurface.new

pipe.create\_swept\_surface(clr, d, @sweep\_opts)

clr.dispose()

pipe.world\_draw(wd)

pipe.dispose()

wd.sub\_entity\_traits.true\_color = c

end

return true

end

return parent\_world\_draw(d, wd)

end

def set\_attributes(d, t)

i = parent\_set\_attributes(d, t)

radius = pipe\_radius\_for\_object(d)

if radius > 0.0

# Set color to yellow

t.color = 2

# and lineweight to .60 mm

t.line\_weight = Ads::LineWeight.line\_weight\_060

end

return i

end

end

When our application is loaded, these two overrules will control the display of lines and circles that have our specific XData attached. But there are times when we want to share the model content – by “exploding” or decomposing the augmented lines and circles into the solid objects we’re using to display them. This will allow the model to be shared with people that are not using our application.

9.2.4 Transforming overrules

For this we need to implement TransformOverrules, through which we will overrule the explode() operation, creating the geometry in much the same way as we did in the DrawableOverrules. Listing 9.19 shows the code where we subclass a TransformOverrule instead of a DrawableOverrule.

Listing 9.19: Defining transform overrules

class LinePipeTransformOverrule < ConcreteClasses::TransformOverrule

# An overrule to explode a linear pipe into Solid3d objects

def initialize

@sweep\_opts = Ads::SweepOptions.new

end

def explode(e, objs)

radius = pipe\_radius\_for\_object(e)

return nil if radius > 0.0

if not e.id.is\_null and e.length > 0.0

# Draw a pipe around the line

start = e.start\_point

endpt = e.end\_point

norm = Ag::Vector3d.new(

endpt.x - start.x,

endpt.y - start.y,

endpt.z - start.z

)

clr = Ads::Circle.new start, norm, radius

pipe = Ads::ExtrudedSurface.new

begin

pipe.create\_extruded\_surface clr, norm, @sweep\_opts

rescue

print\_message "\nFailed with CreateExtrudedSurface."

end

clr.dispose()

objs.add(pipe)

end

super

end

end

class CirclePipeTransformOverrule < ConcreteClasses::TransformOverrule

# An overrule to explode a circular pipe into Solid3d objects

def initialize

@sweep\_opts = Ads::SweepOptions.new

end

def explode(e, objs)

radius = pipe\_radius\_for\_object(e)

return nil if radius > 0.0

if e.radius > radius

start = e.start\_point

cen = e.center

norm = Ag::Vector3d.new(

cen.x - start.x,

cen.y - start.y,

cen.z - start.z

)

clr = Ads::Circle.new start, norm.cross\_product(e.normal), radius

pipe = Ads::SweptSurface.new

pipe.create\_swept\_surface(clr, e, @sweep\_opts)

clr.dispose()

objs.add(pipe)

end

super

end

end

This brings us to the end of creating overrules. We can now draw an overrule and then transform it. What we need to do next is provide some commands for the command line in AutoCAD to enable or disable the overrules.

9.2.5 Commands for the UI

Now all that remains is to implement our commands: entering OVERRULE1 at AutoCAD’s command-line will turn on our overrules, OVERRULE0 will turn them off, and MAKEPIPE will allow us to add XData to lines and circles, defining them as “pipes”.

Listing 9.20 has the overrule1() and overrule0() functions and the function they depend on.

Listing 9.20: the functions for the UI

def overrule(enable)

# Regen to see the effect

# (turn on/off Overruling and LWDISPLAY)

Ar::Overrule.Overruling = enable

if enable

Aas::Application.set\_system\_variable("LWDISPLAY", 1)

else

Aas::Application.set\_system\_variable("LWDISPLAY", 0)

end

doc = Aas::Application.document\_manager.mdi\_active\_document

doc.send\_string\_to\_execute("REGEN3\n", true, false, false)

doc.editor.regen()

end

$overruling = false

$radius = 0.0

def overrule1

begin

unless $overruling

$lpdo = LinePipeDrawOverrule.new

$cpdo = CirclePipeDrawOverrule.new

$lpto = LinePipeTransformOverrule.new

$cpto = CirclePipeTransformOverrule.new

Ads::ObjectOverrule.add\_overrule(

Ar::RXClass::get\_class(Ads::Line.to\_clr\_type),

$lpdo,

true)

Ads::ObjectOverrule.add\_overrule(

Ar::RXClass::get\_class(Ads::Line.to\_clr\_type),

$lpto,

true)

Ads::ObjectOverrule.add\_overrule(

Ar::RXClass::get\_class(Ads::Circle.to\_clr\_type),

$cpdo,

true)

Ads::ObjectOverrule.add\_overrule(

Ar::RXClass::get\_class(Ads::Circle.to\_clr\_type),

$cpto,

true)

$overruling = true

overrule(true)

end

rescue

print\_message("\nProblem found: " + $! + "\n")

end

end

def overrule0

begin

if $overruling

Ads::ObjectOverrule.remove\_overrule(

Ar::RXClass::get\_class(Ads::Line.to\_clr\_type),

$lpdo)

Ads::ObjectOverrule.remove\_overrule(

Ar::RXClass::get\_class(Ads::Line.to\_clr\_type),

$lpto)

Ads::ObjectOverrule.remove\_overrule(

Ar::RXClass::get\_class(Ads::Circle.to\_clr\_type),

$cpdo)

Ads::ObjectOverrule.remove\_overrule(

Ar::RXClass::get\_class(Ads::Circle.to\_clr\_type),

$cpto)

$overruling = false

overrule(false)

end

rescue

print\_message("\nProblem found: " + $! + "\n")

end

end

def makePipe()

begin

doc = Aas::Application.document\_manager.mdi\_active\_document

db = doc.database

ed = doc.editor

# Ask the user to select the entities to make into pipes

pso = Aei::PromptSelectionOptions.new

pso.allow\_duplicates = false

pso.message\_for\_adding =

"\nSelect objects to turn into pipes: "

sel\_res = ed.get\_selection(pso)

# If the user didn't make valid selection, we return

if sel\_res.status != Aei::PromptStatus.OK

return

end

ss = sel\_res.value

# Ask the user for the pipe radius to set

pdo = Aei::PromptDoubleOptions.new "\nSpecify pipe radius:"

# Use the previous value, if if already called

if $radius > 0.0

pdo.default\_value = $radius

pdo.use\_default\_value = true

end

pdo.allow\_negative = false

pdo.allow\_zero = false

pdr = ed.get\_double(pdo)

# Return if something went wrong

if pdr.status != Aei::PromptStatus.OK

return

end

# Set the "last radius" value for when

# the command is called next

$radius = pdr.value

# Use a transaction to edit our various objects

tr = db.transaction\_manager.start\_transaction()

# Loop through the selected objects

for o in ss do

# We could choose only to add XData to the objects

# we know will use it (Lines and Circles, for now)

obj = tr.get\_object(o.object\_id, Ads::OpenMode.for\_write)

set\_pipe\_radius\_for\_object(tr, obj, $radius)

end

tr.commit()

tr.dispose()

rescue

print\_message("\nProblem found: " + $! + "\n")

end

end

add\_commands ["overrule1", "overrule0", "makePipe"]

The makepipe() function prompts the user to select objects and provide the radius to assign, and then iterates through and attaches the corresponding XData to each one. Finally we register our commands. Now we’re ready to test our work.

9.2.6 Executing the code

To put our code through its paces, we need to load our compiled C# module into AutoCAD using the NETLOAD command, and then type RBLOAD at the command-line.

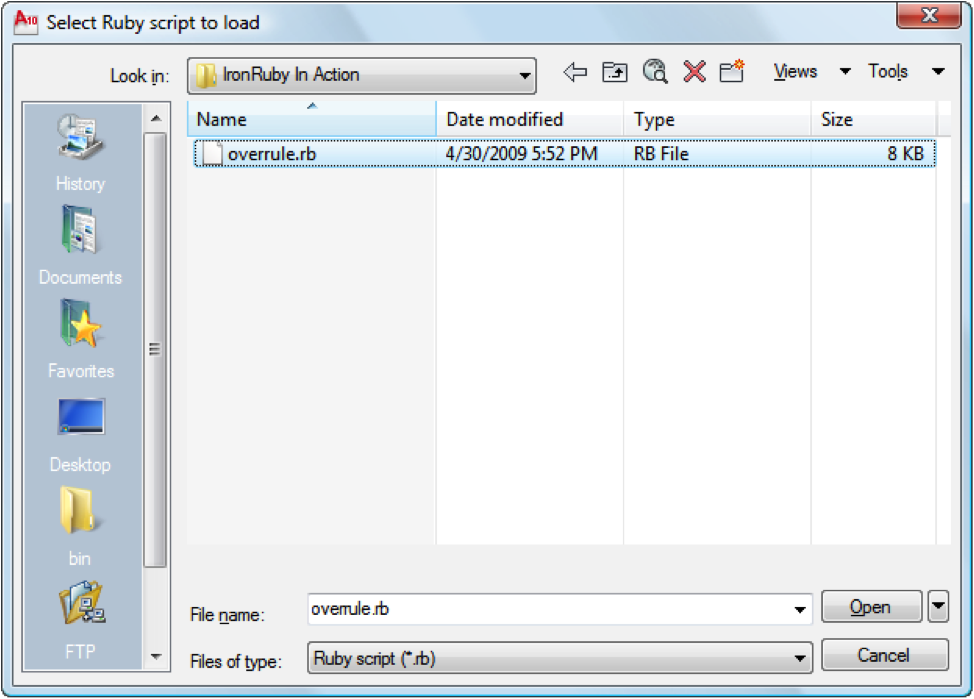


Figure 9.1: Loading the ruby script

Once we select our script, it will be loaded and executed by the hosted IronRuby engine, registering our three commands.

Registered Ruby command: overrule1

Registered Ruby command: overrule0

Registered Ruby command: makePipe

Using standard AutoCAD drawing commands, we can create some simple lines and circles.

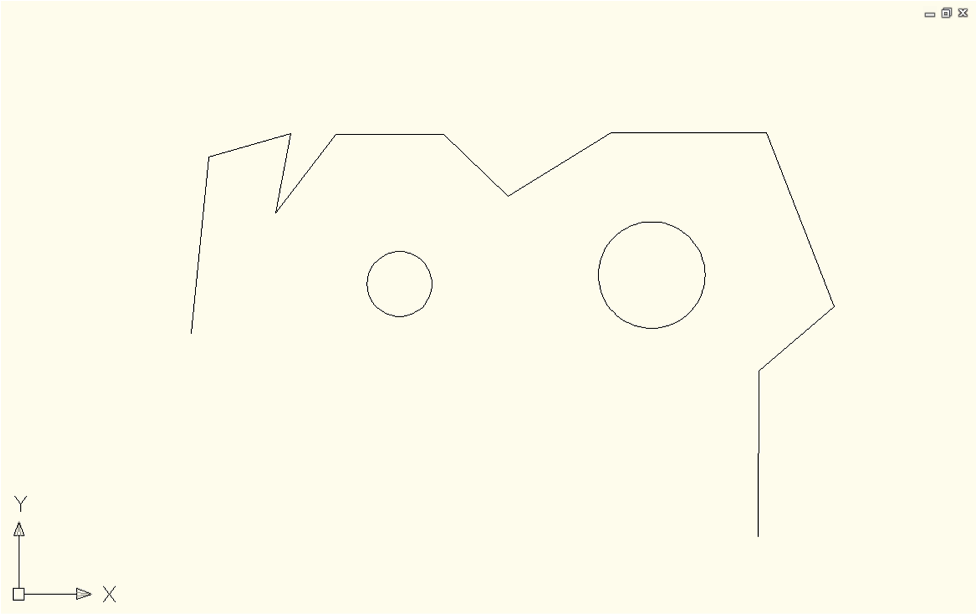


Figure 9.2: Creating some circles and lines

The MAKEPIPE command allows us to assign radii to the various entities, which can then be made visible using the OVERRULE1 command.

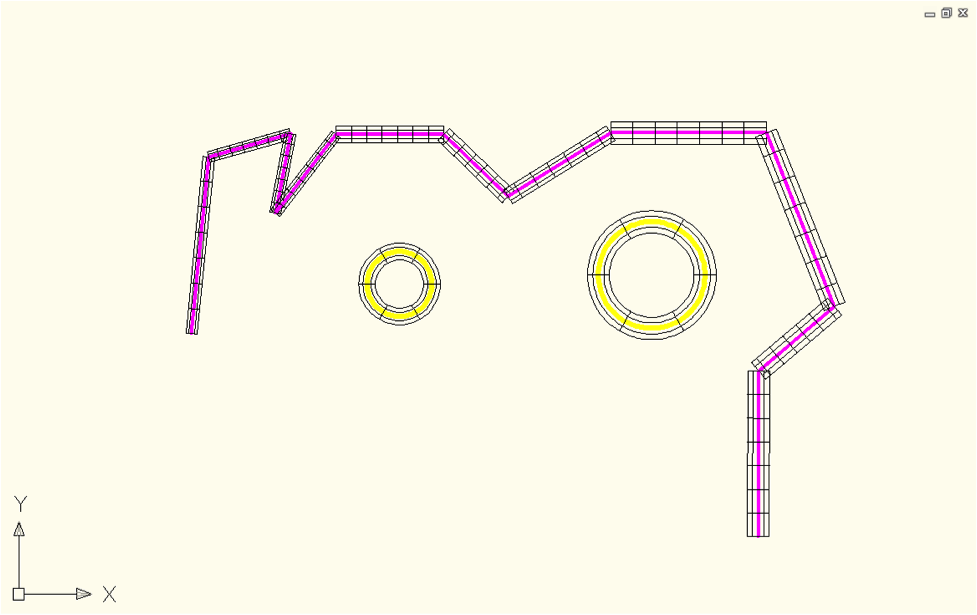


Figure 9.3: Applying the overrule.

We can see this geometry better in a three-dimensional view.

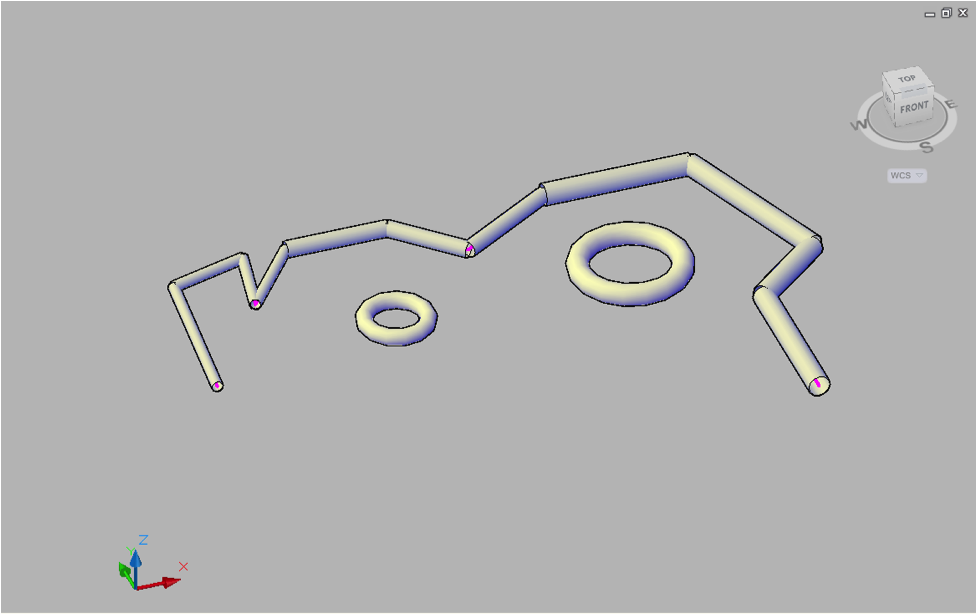
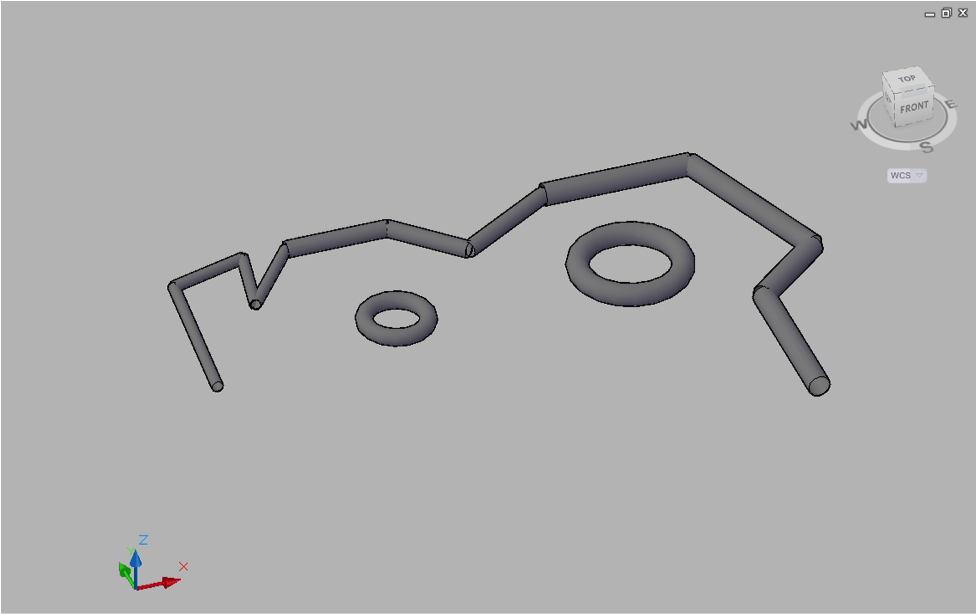


Figure 9.4: The overrule in 3D view.

And when we EXPLODE this geometry, we see the resultant 3D Solid objects.



That’s it for this introduction to using IronRuby in AutoCAD 2010. For more information please visit the Through the Interface blog (<http://blogs.autodesk.com/through-the-interface>).

9.3 Summary

I would like to thank Kean and Thibaut for their contributions to this chapter. That windows forms DSL certainly makes creating those type of GUI applications a lot more fun. More importantly I hope it gave you some insight into a typical ruby workflow and how to go about meta-programming to do more with less.

Kean in turn touched on hosting the DLR in an existing application like AutoCAD and then script that application with IronRuby. I thought that was pretty impressive and although the example was fairly simple it was good for somebody like me who never gets into drawing applications.

In the next chapter we’ll discover how we can extend IronRuby with our own classes as if they were ruby types. We’ll also look at how we can use libraries from other .NET languages inside IronRuby.