Programming with Boo

BOO SPEAK

Hooked on Python's "wrist-friendly" syntax? Enamored of .NET architecture but equally appreciative of C#'s strong typing? Boo offers the best of three worlds. **BY MARTIN STREICHER**

Piumadaquila, Fotolia

Unlike other development platforms, the .NET framework can mix and match code from any number of programming languages. For those who program in .NET, the universal code form known as Microsoft Intermediate Language (MSIL) is a lingua franca. If you can translate your source code into MSIL, you can combine it with, say, Visual BASIC.NET or C# to produce one executable.

Indeed, given such flexibility, developers have adapted many popular languages to .NET. IronPython [1] is a full implementation of Python for .NET, and IronRuby [2] is a proposed .NET-ready implementation of Ruby. Also, you can find Java, Lisp, and Smalltalk for .NET. Moreover, if you don't like any of the existing programming languages, you can create your own. If you can consume source code and produce MSIL, the sky is the limit.

In fact, that's the genesis of Boo [3]. Hooked on Python's sparse, "wrist-friendly" syntax but enamored of the .NET architecture and the strong typing found in C#, developer Rodrigo Barreto de Oliveira set out to combine the best features of both – with just the right amount of Ruby – into something readily suited to iterative development.

In this article, I help you get started with Boo and show you how to use the MonoDevelop IDE to write and test some Boo code [4] [5]. Boo is available through an MIT-style license, which provides broad latitude for using the code and creating derivative works.

Python + Ruby + C# = Boo

Python has minimal syntax and uses white space to connote program structure. Boo follows suit. Listing 1 is a Boo program that reformats text lines to fit within a specific width.

The output of *test("this is a very long string")* is wrapped text with no more than 12 characters per line:

```
this is a*
*very long*
*string
```

As you can see from Listing 1, Boo is very spartan: no semicolons, no braces, no classes (if not needed), and no mandatory variable declarations. Just assign and go. But Boo is simple in form only. Type is enforced both implicitly and explicitly. Formal arguments can be typed, as you can see in the definition of the *wrapLines()* method. Otherwise, type is inferred.

For example, the assignment *lines = []* implies that *lines* is a list. Similarly, *nextBreak = columns* infers that *nextBreak* is an *int*.

Ruby infers type as well. As in Ruby, everything in Boo is an object. The following is an interesting snippet of code from the *Boo Primer* [6]. (Once you build the Boo interactive shell, as described later in this article, you can try this code yourself.)

```
o as object
o = '12'
o = 4
```

Because *o* is instantiated as an object, the subclass of all types, it can reference any other type. This canonical example demonstrates polymorphism. However, if you declare *i* more narrowly at the outset, Boo catches an errant assignment.

```
i as int
i = 4
i = '12'
a = '12'
--------
ERROR: Cannot convert ➘
'string' to 'int'.
```

Ruby and Boo have another language feature in common: Both support duck typing, as in "If it walks like a duck and quacks like a duck, it must be a duck." Or more to the point, if an object looks like a string, acts like a string, and responds to the same methods that an object of type string responds to, then it

Listing 1: Reformatting Text

```
01 import System.Text.
   RegularExpressions
02
03 def getWordBreaks(s as string):
04     return m.Index for m as Match in
       /\b|$/.Matches(s)
05
06 def wrapLines(s as string, columns as
                int):
07
08     lines = []
09
10     nextBreak = columns
11     lastBreak = 0
12     lineStart = 0
13     for wb as int in
        getWordBreaks(s):
14         if wb > nextBreak:
15             line =
                 s[lineStart:lastBreak]
16             lines.Add(line.Trim())
17             lineStart = lastBreak
18             nextBreak = lastBreak +
                           columns
19         lastBreak = wb
20
21     lines.Add(s[lineStart:].Trim())
22
23     return lines
24
25 def test(s):
26     print(join(wrapLines(s, 12),
                  "*\n*"))
```

must be either a string, or a clever enough mimic to pass as a string.

To use duck typing in Boo, use the eponymous type *duck*. Boo forgoes compile-time type checking on variables declared as a duck.

```
fudd as int
daffy as duck
```

```
fudd = 0
daffy = 1
daffy = "quack"
fudd = "be vewy quiet"
ERROR: Cannot convert ➥
'string' to 'int'.
```

Typically, you only have to specify a type when it is to your advantage. For instance, Boo provides method overloading, an added benefit of declaring the type of arguments. Otherwise, do not declare a type; let inference do the heavy lifting.

```
# x in an int and can do math
x = 5
x += 5

# now, x is a string
x = 'hello'
print x.ToUpper()
'HELLO'
```

The most common of programming types have succinct, lowercase names, such as *object*, *int*, *string*, *double*, and *bool*. More complex types are capitalized, as in *List* or *Match*.

Booting Up Boo

To use Boo, you must install the MonoDevelop integrated development environment (IDE), a suite of coding tools designed for C# and other .NET languages. The latest version of MonoDevelop is 1.9.2, a pre-release of the

Boo-dles of Good Ideas

The *Boo Primer* [6] is a well-written introduction to the Boo language. The constructs and syntax of each element are described in detail; you can also learn quite a bit by browsing the sample code provided with the Boo compiler and its test suites. As you study the Boo language, you will start to get a sense of how it works.

Boo offers both a list and an array type. A list can vary in size, and an array is fixed in size. You can use either a list or an array to extract one, some, or all elements, and both can contain heterogeneous elements. Listing 2 demonstrates some similarities and differences between a list and an array.

The code in Listing 2 produces the following:

```
[0, 1, 2, 3, 4]
['hello', 1, true, System.Object]
[10, 'hello']
[1, True]


(0, 1, 2, 3, 4)
(0, 1, 5, 3, 4)
(1, 5)
```

Both the list and array use brackets (*[]*) for indexing and slicing; a list is displayed with brackets, and an array appears with parentheses to differentiate. A slice can be a single element or a bounded or unbounded range. *m[1:3]* includes the element at index *1* and everything up to but not including the third element. *m[:2]* is the set of all elements up to but not including the second to last. An out-of-bounds access in either a list or an array yields an exception.

Interestingly, a string behaves like a range and a list.

```
print (s = List('abcdefghi'))
[a, b, c, d, e, f, g, h, i]
print s[0:4]
[a, b, c, d]
print (s.Add('z'[0]))
[a, b, c, d, e, f, g, h, i, z]
```

Like Perl and Ruby, Boo also offers a hash to store (key, value) pairs.

A *struct* (short for structure) is like a class – a *struct* can have instance variables and methods, but it is stored as a value instead of a reference. A method is an object, too, and it has its own methods. Making each function its own object has practical uses. For example, you can instantly run a function in its own thread by calling the method's own *BeginInvoke()* method. You can terminate the thread by calling the function's *EndInvoke()* method.

Boo also provides a macro named *lock* to grant a thread mutually exclusive access to an object during an operation.

Other macros are provided, too, but the more compelling story is that anyone can create a new macro using any CLI language. Macros save typing, yes, but also reduce errors and compartmentalize sections of code. For example, a *with* macro can transform the wordy sequence:

```
fooInstanceWithReally➥
LongName = Foo()
fooInstanceWithReally➥
LongName.f1 = 100
fooInstanceWithReally➥
LongName.f2 = "abc"
fooInstanceWithReally➥
LongName.DoSomething()
```

… into the more succinct statement …

```
with fooInstanceWith➥
ReallyLongName:
    _f1 = 100
    _f2 = "abc"
    _DoSomething()
```

Boo has other nice features, but the macro, which allows the creation of domain-specific languages, might be its most powerful feature.

Listing 2: List and Array

```
01 l = List(range(5))
02 m = ['hello', 1, true, object]
03 n = []
04 n.Add(10)
05 n.Add('ten')
06 print l
07 print m
08 print n
09 print m[1:3]
10
11 a = array(range(5))
12 print a
13 a(2) = 5
14 print a
15 print a[1:3]
```

forthcoming MonoDevelop 2.0. Binary packages of Mono and MonoDevelop are available for a variety of Linux distributions and other platforms, including Mac and Windows, or you can build the software from source.

To build from source, make sure your Linux machine has the typical suite of development tools required to build Gnome applications.

Some other utilities and development libraries are also required: a Java Runtime, the Bison parser generator, the pkg-config utility, the Pango rendering library, the ATK accessibility library, the Gtk 2.0 libraries, the Curses library, some CLI bindings, and the GNU Library (glib) version 2.0 or greater. For the necessary packages, check your distribution's package manager.

Next, download and unpack the tarballs for MonoDevelop and its dependencies. (See a comprehensive list of components on the Mono site [7].) Nine packages are required (Listing 3).

Pay particular attention to the instructions for GDI+: You must configure the loader to pick up libraries in */usr/local/lib*. If you attempt to run some of the Boo Windows Forms applications and get an error such as *System.DllNotFoundException: gdiplus.dll*, you skipped this step. (See the Mono project website for more information [8].)

A complete build takes about 30 minutes and yields the Mono engine, the *nant* build tool (similar in purpose to *make*), the MonoDevelop IDE, a debugger, and a panoply of Linux and .NET libraries. By default, all of the software is installed in */usr/local* or its subdirecto-

Listing 3: Setting Up MonoDevelop

```
01 # Make a workspace
02 $ mkdir $HOME/mono
03 $ cd $HOME/mono
04
05 $ # Download, unpack, and build all the source code
     packages required
06 $ # Mono
07 $ wget http://ftp.novell.com/pub/mono/sources/mono/
         mono-2.2.tar.bz2
08 $ tar xjf mono-2.2.tar.bz2
09 $ ( cd mono-2.2; ./configure; make; sudo make install )
10
11 $ # GDI+
12 $ wget http://ftp.novell.com/pub/mono/sources/libgdiplus/
         libgdiplus-2.2.tar.bz2
13 $ tar xjf libgdiplus-2.2.tar.bz2
14 $ ( cd libgdiplus-2.2.; ./configure; make; sudo make
      install )
15 $ # Tell the loader to look in /usr/local/lib for libraries
16 $ sudo echo '/usr/local/lib' >> /etc/ld.so.conf.d/win32.
             conf
17 $ sudo ldconfig
18
19 $ # GTK#
20 $ wget http://ftp.novell.com/pub/mono/sources/
         gtk-sharp212/gtk-sharp-2.12.7.tar.bz2
21 $ tar xjf gtk-sharp-2.12.7.tar.bz2
22 $ ( cd gtk-sharp-2.12.7; ./configure; make; sudo make
      install )
23
24 $ # GNOME#
25 $ wget http://ftp.novell.com/pub/mono/sources/
         gnome-sharp220/gnome-sharp-2.20.1.tar.bz2
26 $ tar xjf gnome-sharp-2.20.1.tar.bz2
27 $ ( cd gnome-sharp-2.20.1; ./configure; make; sudo make
      install )
28
29 $ # Mono Tools
30 $ wget http://ftp.novell.com/pub/mono/sources/mono-addins/
         mono-addins-0.4.zip
31 $ unzip mono-addins-0.4.zip
32 $ ( cd mono-addins-0.4; ./configure; make; sudo make
      install )
33
34 $ # The MonoDevelop IDE
35 $ wget http://ftp.novell.com/pub/mono/sources/monodevelop/
         monodevelop-1.9.2.tar.bz2
36 $ tar xjf monodevelop-1.9.2.tar.bz2
37 $ ( cd monodevelop-1.9.2; ./configure; make; sudo make
      install )
38
39 $ # The Nant Build Tool
40 $ wget http://superb-east.dl.sourceforge.net/sourceforge/
         nant/nant-0.86-beta1-src.tar.gz
41 $ tar xzf nant-0.86-beta1-src.tar.gz
42 $ (cd nant-0.86-beta1; make; sudo make install )
43
44 $ # The Mono Debugger
45 $ wget http://ftp.novell.com/pub/mono/sources/
         mono-debugger/mono-debugger-2.2.tar.bz2
46 $ tar xzf mono-debugger-2.2.tar.bz2
47 $ ( cd mono-debugger-2.2; ./configure; make; sudo make
      install )
48
49 $ # Support for the Mono Debugger in monodevelop
50 $ wget http://ftp.novell.com/pub/mono/sources/
         monodevelop-debugger-mdb/\
         monodevelop-debugger-mdb-1.9.2.tar.bz2
51 $ tar xjf monodevelop-debugger-mdb-1.9.2.tar.bz2
52 $ (cd monodevelop-debugger-mdb-1.9.2; make; make install )
```

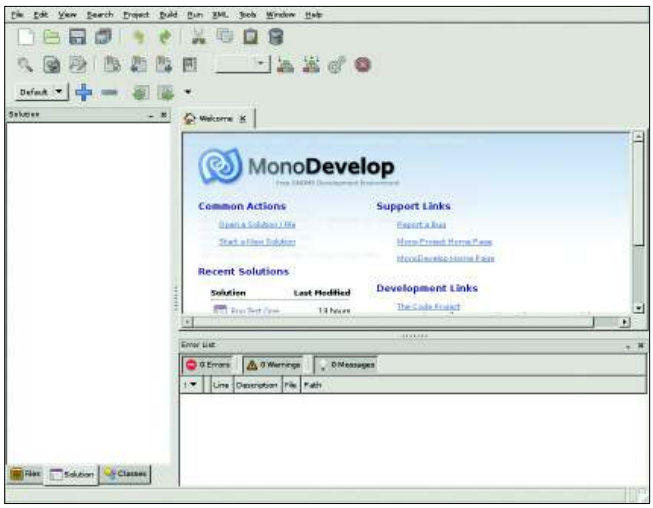

Figure 1: The MonoDevelop main window.

ries. In each command, use *./configure --prefix directory_name* to specify an alternate target, such as */opt/local*.

The next step is to download and build the Boo compiler and the Boo environment for MonoDevelop:

```
$ wget ➥
  http://dist.codehaus.org/boo/distributions/➥
  boo-0.9.0.3203-2-src.zip
$ mkdir boo; unzip ../boo-0.9.0.3203-2-src.zip
```

Before you continue, edit the file *default.build* and change the line <*property name*="*skip.vs2005*" *value*="*False*" /> to read <*property name*="*skip.vs2005*" *value*="*True*" />. This step skips the Visual Studio–specific portion of the Boo build. Now, compile Boo:

```
$ cd boo
$ /usr/local/bin/nant rebuild
$ /usr/local/bin/nant update-bin
$ /usr/local/bin/nant ➥
  compile-tests && ➥
  nunit-console tests/build/*Tests.dll
$ /usr/local/bin/nant install
$ cd ..
```

Now you have a complete working copy of the Boo compiler *booc*. To try it, build and execute one of the example Boo programs found in *./examples*.

```
$ # Compile and run the code
$ booc examples/arrayperformance.boo
$ mono arrayperformance.exe
153.613 elapsed.
250.573 elapsed.
216.785 elapsed.

$ # Interpret the code
$ booi arrayperformance.boo
153.613 elapsed.
250.573 elapsed.
216.785 elapsed.
```

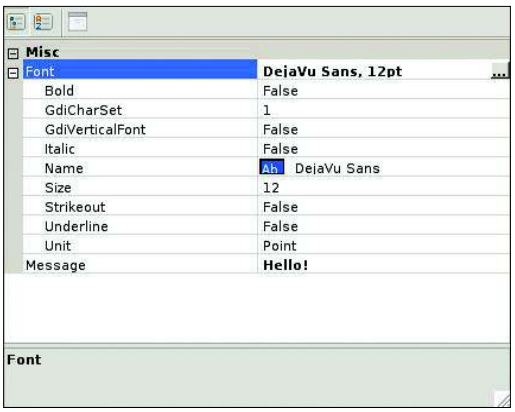

Figure 2: Fonts and Boo – Listing 6 results.

Success! The sample, *arrayperformance.boo*, copies a large array and benchmarks the elapsed time. Also, you can test Boo directly in the Boo interactive interpreter, *booish* (Listing 4).

At long last, you are ready to add Boo support to MonoDevelop (see Listing 5). A link to scripts that build the software for you can be found at the Linux Magazine website [9].

Using MonoDevelop to Create Boo Programs

First, launch the MonoDevelop IDE with *monodevelop*. The main window resembles Figure 1. To create a Boo application, click on *Start a New Solution*, click *Boo*, and select *Empty Project*. Next, pick a name for the project and a location in which to store it and click *OK*. On the next screen, enable *Unix Integration* (optional) and click *OK*. Now you should have a solution in the list on the left.

Here, you need to choose the solution, right-click, and choose *Add | New File…*; next, choose *Empty File* and give the file a name, then click *New*. At this point, you are ready to edit Boo code in the main panel. The editor colors syntax as you type, and it automatically indents on the basis of context. And because you are in Mono, you can build on the .NET frameworks or any framework with a CLI binding, such as Gtk#.

Now enter the code in Listing 6 in MonoDevelop, save the file, and choose *Run | Run*.

Listing 6 produces the output shown in Figure 2. This example also demonstrates Boo classes. The phrase *class PropertyEditor(Form)* defines the class *PropertyEditor* and derives it from the Windows Form's class *Form*. The *constructor()* method is self-evident. The statements *[property(Message)]* and *_message as string* create a setter and getter method named for the *property _message*, which is a Boo string. The *[property(Font)]* statement is similar, although it is a font.

Saving Keystrokes and Sanity

The Boo project page offers recipes for database access, graphics solutions, and more. Boo developers socialize in a special-interest Google group, in an IRC channel dedicated to the language, and in a mailing list and wiki. Currently, the team is focused on improving Boo support in MonoDevelop.

If you are on Windows, Linux, or any platform with Mono and you are tired of typing braces and casting values, check out Boo. It has the best of at least three worlds. ■

Listing 4: Working with booish

```
01 Welcome to booish, an interpreter for the boo programming language.
02 Running boo 0.9.0.3203 in CLR v2.0.50727.1433.
03
04 Enter boo code in the prompt below (or type /help).
05 >>> print "Hello, world"
06 Hello, world
```

Listing 5: Adding Boo

```
01 $ # The next two steps are required until the boo makefiles used in
02 $ # the previous step are fixed.
03 $ sudo mkdir -p /usr/local/lib/boo
04 $ sudo cp /usr/local/lib/mono/boo/*.dll /usr/local/lib/boo
05
06 $ wget http://ftp.novell.com/pub/mono/sources/monodevelop-boo/
          monodevelop-boo-1.9.2.tar.bz2
07 $ tar xjf monodevelop-boo-1.9.2.tar.bz2
08 $ ( cd monodevelop-boo-1.9.2; ./configure; make; sudo make install )
```

Listing 6: Trying Out Boo Classes

```
01 import System.Windows.Forms from
   System.Windows.Forms
02 import System.Drawing from System.
   Drawing
03
04 class PropertyEditor(Form):
05     def constructor([required]
                       obj):
06         grid = PropertyGrid(Dock:
   DockStyle.Fill,   SelectedObject:
   obj)
07         Controls.Add(grid)
08
09 class Options:
10     [property(Message)]
11     _message as string
12
13     [property(Font)]
14     _font as System.Drawing.Font
15
16 options = Options(Message:
   "Hello!", Font: Font("Lucida
   Console", 12.0))
17 editor = PropertyEditor(options)
18 editor.ShowDialog()
19 print(options.Message)
```

INFO

[1] IronPython: *http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython*

[2] IronRuby: *http://www.ironruby.net/*

[3] The Boo Project page: *http://boo.codehaus.org/*

[4] The Mono Project page: *http://mono-project.com/*

[5] The MonoDevelop IDE: *http://monodevelop.com/*

[6] *Boo Primer*: *http://boo.codehaus.org/Boo+Primer*

[7] A comprehensive list of source packages for Mono: *http://ftp.novell.com/pub/mono/sources-stable*

[8] Troubleshooting guides for missing libraries: *http://mono-project.com/DllNotFoundException* and *http://mono-project.com/Config_DllMap*

[9] Listings for this article: *http://www.linux-magazine.com/resources/article_code*