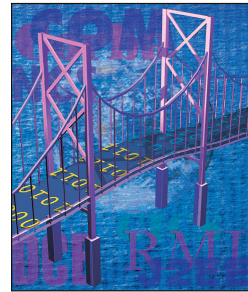
Ruby Extensions



Steve Vinoski • *IONA Technologies*

Last time, I reviewed Maik Schmidt's *Enterprise Integration with Ruby*[1] as a means of exploring dynamic languages' applicability to middleware integration projects. Using languages such as Ruby for these projects is straightforward when developers can create pure dynamic language applications that access preexisting services, even though such services (often considered "legacy") are typically implemented in "traditional" middleware languages such as Java, C++, or C. This approach works reasonably well because the dynamic language applications typically reside in separate address spaces from the services they use, accessing the services only through avenues that guarantee separation, including database drivers and network connections.

Yet, not all enterprise integration projects involve such clean separation. Are dynamic languages like Ruby still useful in situations that require directly coupling the dynamic code to the legacy code? Lately, in my own work, I've been exploring what it takes to cleanly integrate Ruby into an existing C++ middleware system. In this column, I describe some of the issues I've run into along the way and detail the approaches I've found to work best.

Java is Easy

First, let me state that I won't explore the integration of dynamic languages with Java in this column because the process is pretty simple. The Java Virtual Machine (JVM) is steadily evolving into a multilanguage platform, partly in response to competition from Microsoft's Common Language Runtime (CLR) and partly due to demand from dynamic language aficionados. Multiple dynamic language implementations already exist on the JVM, including JavaScript, ECMAScript for XML (E4X), Jython, BeanShell, Groovy, and JRuby. In fact, Java 6 will include the Mozilla Rhino JavaScript implementation.

Integrating dynamic languages with Java is relatively easy because at runtime, the languages all share the same VM under the covers, which lets dynamic languages easily call into and share data with Java code and vice versa. Several Java Specification Requests (JSRs) — specifically, 223 (scripting engine), 241 (the Groovy scripting language), 274 (the BeanShell scripting language), and 292 (additional VM bytecode for dynamic language support) — make it clear that integrating dynamic languages with Java is growing in importance and will thus get even easier over time.

Two Approaches

Unfortunately, integrating C-based dynamic language implementations with C++ and C applications is more challenging because of the diversity of approaches and application architectures involved. Unlike Java, such systems lack a unifying VM underneath them. Furthermore, middleware systems normally use event loops, multiple threads, and other such features, which can create integration nightmares because each application typically assumes that it has complete control in those areas.

Combining a chunk of C++ or C middleware with a dynamic language to create a cohesive application normally means wrapping the middleware with the dynamic language. Under such a setup, developers work directly with the dynamic language, and the middleware hidden underneath becomes essentially an implementation detail. Layering a C-based dynamic language implementation over C++ or C middleware generally requires "glue code" that resides between the dynamic language and the middleware. Developers can create this code either manually or using code-generation tools.

One of the best-known code-generation tools for automating the integration of C++ or C code into a dynamic language is the Simplified Wrapper and Interface Generator (SWIG; www.swig.

org). A developer creates an interface description file that declares the C++ or C functionality to export to the dynamic language, from which SWIG tools generate code that ties that functionality into the conventions and contracts expected by the dynamic language. Such code is then typically compiled into library modules that the dynamic language runtime can load on demand. SWIG supports a wide variety of dynamic languages, including Ruby.

Unfortunately, SWIG is not a panacea. The code it generates can easily suffer from impedance-mismatch problems, not unlike those encountered when exporting Java or C++ objects directly as Web services.[2] The C++ or C functions that SWIG makes available directly to the dynamic language environment can often be a poor fit due to style or granularity issues.

For example, while writing this column, I was playing with the IOKit framework (http://developer.apple.com/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/) on Mac OS X 10.4, which lets programs access computer power information. Through the framework, programs can determine, for example, whether the system is running on AC power or on battery, and how much charge is left in the battery. The AC cord on my PowerBook G4 often slips out just enough so that the machine starts running off the battery even though I think it's still plugged in, and unfortunately I usually don't notice until I start getting low-battery warnings. I therefore decided to write an application to pop a notification window onto the screen if the machine switched to battery power. Having no desire to implement this entire application in C++, C, or Objective-C, I decided to write it in Ruby and use SWIG to make the framework functions available at the Ruby level.

The portion of the IOKit framework that I needed had only three C functions, so I first tried to get SWIG to simply process the entire C header file that declared those functions. Unfortunately, the SWIG parser choked on a fairly straightforward typedef for a function pointer, so I then tried to explicitly export the three functions. That appeared to work, except that once I tried to use the functions from Ruby, I realized that the function input and return values were opaque C data types that could be accessed only via other C functions. At that point, I had two choices:

- I could use SWIG to also export the data-manipulation and access functions necessary to handle the opaque data types. The problem with this approach is that the granularity of the exposed functions didn't fit well with the Ruby language. Ruby is strongly object-oriented, but by themselves, these functions didn't resemble natural Ruby classes or objects. Furthermore, using the functions properly requires explicit memory-management calls, which are, of course, anathema to dynamic language users.
- I could try to use SWIG's typemap facility to effectively inject handwritten code snippets into the generated code to map between the framework types and Ruby types. Unfortunately, the conversions that were possible with typemaps didn't seem suitable for solving this particular problem.

Even if I could've used typemaps, I would've had to write custom glue code, so in the end I chose to avoid SWIG altogether and write my own Ruby glue code in C.

I also tried SWIG for my middleware project, which involves a much larger C++ framework, and there too, I encountered similar problems. In that case, I again entirely avoided SWIG by writing my own glue code instead. It seems SWIG would work well only if the underlying code happened to fit into the dynamic language's style, which I expect would be uncommon. If you try SWIG, perhaps you'll have better luck than I did, but overall, I recommend designing and implementing your own glue code instead.

The resulting productivity gains that Ruby can offer your integration projects are well worth the effort.

Writing Extensions

Fortunately, writing glue code or extensions in C++ or C for Ruby isn't that difficult. The hardest part is finding detailed descriptions of the Ruby C API functions that you'll need to invoke from within your extension. *Programming Ruby*,[3] a popular book informally called "the pickaxe" because of its cover art, documents many of the extension API functions, as well as the general approach to writing Ruby extensions in C. If you intend to write your own extensions, I recommend keeping a copy of this book handy.

Let's assume you're writing an extension to let Ruby applications directly access your C++ or C middleware libraries. Many of the modules that make up the installed Ruby package are written not in Ruby but in C, as they often have to access underlying operating system facilities. Your middleware extension will provide a new Ruby module that wraps the middleware functionality — just as the aforementioned Ruby modules wrap operating system functionality. Within your Ruby applications, you'll use your middleware module the same as

you'd use any other Ruby module, regardless of whether it's written in Ruby, C, or some other language. Even if the pickaxe doesn't document a particular Ruby API function you need, your extension module will necessarily use many of the same functions that some of the installed Ruby modules use, so you can also use the Ruby source code itself to figure out what any particular function does.

The Ruby `mkmf` module makes building an extension quite easy. You simply create a Ruby file named `extconf.rb` with the following contents:

```
require 'mkmf'
create_makefile('myextension')
```

where the `myextension` string refers to the name of the Ruby module you wish to produce. You then execute the following command:

```
ruby extconf.rb
```

to create a Makefile. You can then build your extension by simply running "make." You can add quite a few optional directives to your `extconf.rb` file to customize the build — to set include paths, for example, or to verify the existence of certain libraries or functions. (See the `mkmf` module documentation or the pickaxe for more details.)

Executing `extconf.rb` builds all C++ and C source files found in the current working directory into the Makefile. Running "make" then creates a shared library or dynamically linked library (DLL) that Ruby loads into your application when you require the module. For a module named `myextension`, Ruby expects to find a C function named `Init_myextension` in the shared library. Upon loading the shared library, Ruby invokes this function to initialize the extension. Within this function, you call the necessary Ruby API functions to set up the Ruby modules, classes, and functions that your extension provides.

The hard part, of course, is designing the functionality to export to the Ruby layer and writing the code to implement it. Because of the impedance mismatch issues I described earlier, I recommend that you actually design what you want to export to the Ruby layer, rather than just relying on mechanical code generation to do it for you. Returning to my Mac IOKit framework example, rather than blindly exporting the C functions I needed to access my laptop's AC power and battery information, along with the various data-type support functions they in turn require, I chose to write one C function to wrap them. Making this the only function available at the Ruby layer kept many unnecessary low-level details out of my Ruby code.

For my Ruby/C++ middleware-integration project, I'm employing a multilayered approach in which Ruby applications use a module written in Ruby. This module in turn wraps a module written in C, which directly accesses my middleware system. This layered approach offers several benefits:

- The application layer resides at a Ruby-to-Ruby, rather than a Ruby-to-C, boundary. This simplifies the application-interface development, as I can implement much of it in Ruby, which is much easier to work with than C.
- The Ruby module encapsulates the Ruby-to-C boundary, thus hiding it from direct application access. This lets me move functionality across that boundary (from C to Ruby or vice versa) without affecting the application interface or breaking existing applications.
- Because the Ruby-to-C boundary is hidden, the C code needn't completely solve the impedance-mismatch problem on its own. Instead, part of the impedance mismatch can be solved in the C layer and part of it in the Ruby module. The Ruby module can contain non-idiomatic code if necessary to deal with the C layer without adversely affecting Ruby applications.

SOA developers might notice that this design process exactly matches what's required for making the functionality for a particular service available to applications in a SOA network. Trying to create services by directly exporting objects from the underlying technology is normally a bad choice because it lets too many implementation details show through, and forces them onto service consumer applications. There's no getting around the fact that SOA developers must engage in proper service design, considering issues such as encapsulation, technology boundaries, coupling, and cohesion.[4]

In all, writing Ruby extensions for middleware integration is relatively straightforward. The idea can seem daunting at first, but between the pickaxe, online documentation and examples, Ruby community lists such as the `comp.lang.ruby` Usenet group, and the Ruby source code itself, it doesn't take long to get up to speed. The resulting productivity gains that Ruby can offer your integration projects are well worth the effort.

References

1. M. Schmidt, *Enterprise Integration with Ruby*, Pragmatic Bookshelf, 2006.
2. S. Vinoski, "RPC Under Fire," *IEEE Internet Computing*, vol. 9, no. 5, 2005, pp. 93–95.
3. D. Thomas, *Programming Ruby*, second ed., Pragmatic Bookshelf, 2005.
4. S. Vinoski, "Old Measures for New Services," *IEEE Internet Computing*, vol. 9, no. 6, 2005, pp. 72–74.

Steve Vinoski is chief engineer for IONA Technologies. He's been involved in middleware for more than 17 years. Vinoski has helped develop middleware standards for the Object Management Group (OMG) and the World Wide Web Consortium (W3C). Contact him at vinoski@ieee.org.