Xampl

Bob Hutchison

24th October 2009

Contents

Part I

Introduction

Chapter 1

Installing Xampl

Part II

Xampl by Example

Over the last ten or eleven years I've taught a bunch of programmers how to use xampl. It normally takes about 45 minutes to get them going… and then an extended period of intermittent answering questions and making suggestions. To do this I took advantage of the shared context of a project we both were working on.

Xampl has never been documented, though several attempts have been made. This time I'm serious.

I think teaching technique won't translate very well to print and example code. I suppose I could try simulate this and choose some well understood domain, like weblogs, and work that through… and hope that'd work. It might if I was sitting beside you, but I'm not. And I have to do something, so I'm going to take a bunch of passes at writing a progressively more complex 'Hello World' program. A bit of a stretch, but…

Chapter 2

Hello World

2.1 Hello World, Take One

2.1.1 The Experiment

Let's see if we can get a start with xampl by using it to print hello-world on the screen. Then I'll explain what happened.

You'll need to create a directory somewhere in which to place the files associated with this project. I'll be defining path names using `.` to mean this directory, so you might want to open a terminal and change directories to it. The source for this example is distributed in the xampl gem and is located at `<path to the xampl gem>/docs/intro/example1` which will likely be read-only and so should be copied someplace else[1].

Xampl works by inspecting a collection of XML files and deriving a class model from what it finds. We'll only need one XML file right now:

```
<hello-world/>
```

Figure 2.1: ./xml/hello.xml

To make this happen and generate the xampl code, run:

Are you in the right directory?

```
[working] >> xampl-gen
comprehend file ./xml/hello.xml
WRITE TO FILE: ./xampl_generated_code/XamplAdHoc.rb
```

[1]If you are lazy, which is possibly one of the very good reasons you are interested in xampl, then you can simply untar the file `<path to the xampl gem>/docs/intro/examples.tgz` (or unzip `<path to the xampl gem>/docs/intro/examples.zip`) to someplace convenient.

That's how I'll show a command line session—a mix of commands (with the `[working] >> prefix` like line 1) and the rest output.

`Xampl-gen` will print a line like line 2 for every xml file it finds, meaning that xampl is have a look at it and analysing it. By default `xampl-gen` will analyse all files that match the pattern `./xml/**/*.xml`[2]. The xml files *must* be well-formed and xampl-gen will choke if they are not.

Xampl-gen will write something like line 3 for each file it generates. In this example there is only one, and it is called XamplAdHoc. This is xampl-gen's default output file, and we haven't told it anything better, so try to suppress your gag-reflex until I show you how to deal with this. But, maybe to calm you down a bit, it is considered *very* bad form to allow xampl to generate code into this file for production use.

For the moment, lets just ignore what's in the XamplAdHoc.rb file. It's a little frightening.

To do something with this, let's run this short little ruby program:

```
1 require 'xampl_generated_code/XamplAdHoc.rb'
2
3 Xampl.transaction("example1") do
4   h = HelloWorld.new
5   puts h.to_xml
6 end
```

Figure 2.2: ./example1.rb

And this is what we get when we run it:

```
1  [working] >> ruby example1.rb
2  <hello-world/>
```

And by golly! I see a hello world right there on line 2.

How'd that happen? The file we just generated was included on line 1. In line 3 we start a transaction, called example1. These transactions are necessary, but in this example doesn't buy us a lot, it is mostly just overhead[3]. For now just remember that when you create or change a xampl object you have to be in a transaction or a Xampl::UnmanagedChange exception will be raised and that involves a rollback. I'll be covering transactions in pretty thorough detail, and there is a prolonged discussion in chapter 4.1 on page 16. Line 4 creates a new instance of the class `HelloWorld`. Line 5 calls the `to_xml` method and prints the result.

[2]all files in ./xml and and all sub-directories for files with the extension `.xml`

[3]Yes, I know... for such a simple program this seems like gross overkill. It is I suppose, but it isn't often that xampl is used on simple programs like this.

HelloWorld is a class generated by xampl. The name is derived from the XML element name using rules described in 3.1 on page 15.

2.1.2 What Did xampl Do?

You can look in the generated code yourself and you'll find that one class and one module were generated and a surprising 150 or more lines of code. The class has 20 methods, and the module has 7. The module and none of the methods are interesting to us right now, all of them will be soon. And all for this near-trivial example.

The class is called `HelloWorld`. It mixes in two modules: `Xampl::XamplObject`, and `Xampl::XamplWithoutContent`, and inherits nothing. The mixin `Xampl::XamplObject` means that his is a xampl object, and it carries a bunch of methods, including the `to_xml` method used in the example. The mixin `Xampl::XamplWithoutContent` means that `HelloWorld` has no content and no references to any other xampl managed objects (like an empty XML element).

2.1.3 Summary

- basic structure of a xampl project on disk
- basic usage of the xampl-gen program
- a simple example of using the generated code
- xampl has some kind of way of dealing with name incompatibilities between XML and Ruby
- xampl creates a Ruby class for each XML elements discovered

2.2 Hello hello?!

2.2.1 The Experiment

Carrying on from the previous experiment... That was nice but it could only say hello to the world, and then only if we squinted. What if we wanted to say hello to more than just the world? What if we wanted to have objects that behaved differently? What if we wanted to say it nicer? What if we wanted to get rid of that horribly ugly XamplAdHoc.rb file and actually put the generated code in modules?

The code for this experiment is in the directory: `<path to the xampl gem>/docs/intro/example2`

```
<hello who='world'
       xmlns='com.xampl.intro.example2'/>
```

Figure 2.3: ./xml/example2.xml

Okay, so lets look at this xml file, Figure 2.3,

There are two important differences between this xml file and the one for the first example:

- there is an attribute, who, associated with the element hello (no more hello-world). The value of the attribute is just for example, it does not find its way into the generated code, it could have been anything including ”.
- There is a namespace defined, in this case com.xampl.intro.example2

As soon as we have a namespace defined, then we can map them to Ruby modules. This is done in a ruby file called `project-generator.rb`. It is always called that and is located in the directory from where the `xampl-gen` command is run. This is what it looks like Figure 2.4 on page 11. This file is loaded by the xampl-gen command if it exists. It re-opens[4] the class `ProjectGenerator` (line 1) and re-defines the method `resolve-namespaces` (line 3). An array of arrays is expected. What line 15 is doing is saying that the namespace com.xampl.intro.example2 should be mapped to the Ruby module `Example2`, and when writing XML xampl should prefer the prefix ex2 to identify the namespace.

So run xampl-gen and you should see something like:

```
[working] >> xampl-gen
comprehend file ./xml/hello.xml
WRITE TO FILE: ./xampl_generated_code/Example2.rb
```

You'll note that the XamplAdHoc.rb file is gone, replaced by Exampl2.rb. Xampl will place all code generated for a module into a single file named after the module. If you open up the generated Ruby file you'll see that the code is now inside the module Example2.

So, lets do something with this. Have a look at this Figure 2.5 on page 13 and its output.

So what's happening here? Line 1 requires the generated Ruby code, line 3 includes the module `Example2` so we don't have to keep specifying the module

[4]xampl re-opens classes routinely and profits greatly by the technique. There's a term, *monkey-patching,* applied to this technique that is intended to be derogatory, but it isn't taken that way in the Ruby community.

```
[working] >> ruby example2.rb
haven't set the who attribute yet...
<ex2:hello xmlns:ex2="com.xampl.intro.example2"/>
Hello hello?!

have set the who attribute to 'world'...
<ex2:hello who="world" xmlns:ex2="com.xampl.intro.example2"/>

<ex2:hello who='world'
           xmlns:ex2='com.xampl.intro.example2'/>
Hello world!

there are two Hello things now...
Hello world!
Hello there!

print an array of the two Hello things...
Hello world!
Hello there!

inspect an array of the two Hello things...
[ <ex2:hello who='world'
             xmlns:ex2='com.xampl.intro.example2'/>,
 <ex2:hello who='there'
            xmlns:ex2='com.xampl.intro.example2'/>]
```

```
class ProjectGenerator

  def resolve_namespaces
    # any array of arrays
    # each sub-array:
    #      0: a string or an array of strings, containing xml namespaces found
    #           in the example xml files an empty string is the default namespace
    #      1: a ruby Module name (get the character cases right)
    #      2: a namespace prefix used when writing xml, optional. A generated
    #           prefix will be used otherwise.

    #[]

    [
            [ 'com.xampl.intro.example2', 'Example2', 'ex2']
    ]


  end

end

```

Figure 2.4: ./project-generator.rb

name. Line 5 re-opens the class `Hello` and defines a `to_s` method. Line 11 starts one of those transactions again.

Line 12 creates a new instance of `Hello`, `h`, and prints it's XML and string representation on lines 15-16 (see lines 3-4 of the output listing). Line 18 sets the value of `h`'s who attribute to '`world`' then prints some stuff resulting in output lines 6-11.

Line 25-28 creates another, different, instance of `Hello`, `h2`, and an array, `all`, that contains `h` and `h2`. Lines 30-37 generate the output lines 17-25.

There's a new method being used called `pp_xml`. It is similar to `to_xml` in that it prints XML, but it does in a prettier way[5]. There is another significant difference that we'll get to when we talk about persistence.

2.2.2 What Did xampl do?

Pretty much the same thing as the first experiment. Except that the file is named after a module, and the generated code is in that module. There is also

[5]the 'pp' in `pp_xml` stands for pretty-print.

code to support an attribute.

2.2.3 Summary

- how to map XML namespaces to Ruby modules
- how an XML attribute maps to a Ruby instance variable
- that there can be more than one instance of xampl objects created
- the values taken by instance variables are in no way constrained to the values found in the XML files the code was generated from[6]
- there is a xampl build-in method, `pp_xml`, that prints xml from a xampl object in a prettier way than that of `to_xml`

[6]This has proved to be incredibly difficult for some programmers to grasp. The xml files used to generate xampl objects are ***not*** schemas, they are examples.

```ruby
require 'xampl_generated_code/Example2.rb'

include Example2

class Hello
  def to_s
    "Hello #{ who ? who : 'hello?' }!"
  end
end

Xampl.transaction("example2") do
  h = Hello.new

  puts "haven't set the who attribute yet..."
  puts h.to_xml
  puts h

  h.who = 'world'

  puts "\nhave set the who attribute to 'world'..."
  puts h.to_xml
  puts h.pp_xml
  puts h

  h2 = Hello.new
  h2.who = 'there'

  all = [ h, h2 ]

  puts "\nthere are two Hello things now..."
  puts h
  puts h2

  puts "\nprint an array of the two Hello things..."
  puts all
  puts "\ninspect an array of the two Hello things..."
  puts all.inspect
end
```

Figure 2.5: ./example2.rb

Part III

Programming Manual

Chapter 3

Code Generation

3.1 Name Generation

Chapter 4

Persistence

4.1 Transactions

4.2 Repositories

4.3 Persistence Managers