XML Driven Classes in Python

Eric van der Vlist (vdv@dyomedea.com)

XML Driven Classes in Python

Open Source Convention

July 2004

Copies : http://dyomedea.com/papers/2004-OSCON/

THE OLD RIVALRY BETWEEN CODE AND DATA

A short history of my personal grasp at the rivalry between code and data

To put XML data driven classes into perspective, I'll start giving a short history of my personal experience with the difficult relationship between code and data. This history is personal, not based on historical researches and covering only two and a half decades, however, I hope that it will give a generic enough view point on where data driven classes come from and which spot they fill.

At the beginning, the code owned the data

When I first started to program (at school, back in the late 70's), my focus was on code : there was code and eventually some data defined in the code itself or attached to the code. The data belonged to the code: it was written (actually punched on paper cards) in the code for that code and had existence apart from that code.

Then data came to existence under the shelter of the code

In the early 80's I wrote and used program for scientific applications: I was among the firsts to measure the increase of CO2 in the atmosphere in 82. This was done on an Apple][in basic an assembler. Data had then an existence by itself (the result of a measurement exists by itself independently of any application) but the format in which it was stored was proprietary and opaque and you needed a program written for that purpose to read that data.

Data gained its emancipation through databases

I moved then to real time computing and electronic research and development where data wasn't munch of an issue and when I moved back to look at data management in the late 80s, the big thing was relational databases.

Relational databases are about giving data a full emancipation: you can access data without writing a single line of code and you can join anything to anything else at query time.

The only downside was that you needed to bend your data to make it fit into tables.

Code tried to take control through objects

Databases had become immensely popular, but code hasn't said its last word and came object oriented programming that claimed that code and data were intimately linked that you couldn't separate them and had to describe and store them together.

Amicable divorce between data in databases and object oriented code

Both code and data went along pretending they'd won and that led to an amicable divorce where data is often stored in relational databases separately from the code and fetched when we need it to be temporarily united with code within ephemeral objects.

XML (and RDF) let data become still more emancipated

After ten years or so of such a divorce, XML came along in my radar screen in late 90s and it became rapidly clear that this wasn't going to re-conciliate code and data but that on the contrary, data was getting still more emancipated to the point where document and data heads could look at the same XML document and see without being totally wrong either a plain text document with a bunch of annotations or a hierarchical set of data.

XML is thus the ultimate episode of the emancipation of data : data made readable by itself, without the need of databases and leaving, like that has always been the case with human spoken or written documents, as many interpretations as readers.

If you take it outside of its Semantic Web landscape and focus on its data model, you can consider RDF as a way to split data that were stored in a same row of a relational table as atomic assertions to gave them more freedom.

Both XML and RDF are thus going in the direction of further emancipating data from code.

Reconciliation through data binding

Data emancipation has a lot of huge benefits often referred at as “loosely coupling”, but when code needs to manipulate data you still need to re-conciliate code (which, these days, is most of the time object oriented) with your emancipated data.

One approach, known as “data binding”, consists in automatizing the transfer of information from XML documents into systems of objects.

The most common approach (known as JAX-B in the Java world and natively available in .Net) consists in creating classes from a W3C XML Schema that contains the serialization and deserialization code to read and write corresponding XML documents.

Another approach consists in using reflexion to map information between XML

documents and existing classes.

Data driven classes : the data is in control

Data-binding is only about automation

Data binding builds bridges between data and code keeping both level. It is the automation of a reconciliation process that was made by hand since the early days of databases. Although very useful, data binding doesn't change the nature of the relation between code and data.

Data driven classes give the control to the data

My first perception of how things might move along came in OSCON last year with the presentation “Data-Driven Classes in Ruby” by Michael Granger and David McCorkhill (http://conferences.oreillynet.com/cs/os2003/view/e_sess/4100).

Not specific to XML

One thing to note is that this approach isn't specific to XML at all. In the OSCON 2003 presentation, XML came along only a couple of time at the same level than RTF, PDF or YAML and most of the examples were shown against relational databases.

Not new

The other thing to note is that, although I discovered it last year, the concept isn't that new and I had been using XML data driven classes in Python before listening to that talk. If I think we need to empathize this type of programming, that's because I had failed to realize how these APIs were different from data binding APIs.

Requires dynamically typed languages

The third general comment about data driven classes is that to be data driven in the sense of the definition we'll be giving next, they rely on dynamically typed languages such as Ruby or Python that can dynamically create classes from the structure of the data itself.

What are "data-driven" classes?

(Bullet points hamelessly borrowed from http://conferences.oreillynet.com/presentations/os2003/granger_mccorkhill.pdf)

All classes are data-driven in some sense.

A class that would adapt its behavior from external data wouldn't need that data and any class is, to some extend, data-driven.

Data-driven classes are classes defined by dynamic information.

What makes “data-driven” class different is that these classes modify or even create their behavior from data and that they are not entirely defined by their source code but also by dynamic information.

Why apply data driven classes to XML?

Because XML is self documented

The reason why XML is such a good candidate to use data driven classes is that XML is self documented : when you open an XML document using a reasonably well designed vocabulary, you can get an idea of what it is about without reading any documentation.

If it's self documented for us, why not for a program?

Furthermore, if you look carefully at the nature of this self-documentation through angle bracket based annotation, you'll see that there isn't much room for interpretation at the syntactical level and that it should be easy to automate : if XML is self documented for us why wouldn't it be self documented for a program?

Other XML data driven classes for Python

There are many other XML data driven in Python, and here are some I am aware of.

Objectify (David Mertz, http://gnosis.cx/download/gnosis/xml/objectify)

Objectify is the XML data driven class implementation that I have been using before being aware of its "data driven" nature. Very straightforward to use.

Anobind (Uche Ogbuji, http://uche.ogbuji.net/tech/4Suite/anobind/)

ElementTree (http://effbot.org/downloads/#elementtree)

xmltramp (Aaron Swartz, http://www.aaronsw.com/2002/xmltramp/)

TRAMP (Aaron Swartz, http://www.aaronsw.com/2002/tramp) – RDF data driven classes.

XML Driver: my implementation

Why yet another implementation?

The list was shorter at that time

That's a weak excuse, but the list of implementations of XML data driven classes was shorter at that time: Uche hadn't published his Anobind yet and I wasn't aware of Aaron's TRAMP and xmltramp.

I wanted to try by myself

To be honest, the main reason is that after having attended to the “data driven classes in Ruby” talk, I was very excited and I wanted to try to use the concept by myself.

Emphasis on the data driven nature

I also wanted to put the emphasis on the “data driven” nature of this API instead of presenting it as a parsing or binding tool for XML.

Just in time binding

The last point is as dogmatic as the previous one, but I wanted to take the principal of “late binding” between systems that is one of the strengths of XML to its limits and differ the creation of the class that represents an XML information item to the moment when the application does its first attempt to access to this information item.

A couple of examples

Reading a RSS 1.0 document (simple but not RDF cosher)

```
#!/usr/bin/env python

import XmlObject

def main():
    feed=XmlObject.XmlObjectDocument()
    feed._ParseUri
("http://xmlfr.org/actualites/breves/breves.rss10")
    print "XMLfr wire:\n"
    for item in feed.RDF.item:
        print ("%s" % item.title._value())

if __name__ == '__main__':
    main()
```

The main trick to learn if you want to use my API is that after the document has been parsed, each XML element and attribute is readable as an object from a class that is generated by the API the first time you've tried to access to an element or attribute with the same name.

If it's a leaf node, the value of this element or attribute is available through the "_value()" method.

If not, the attributes and sun elements are available through their local names.

In this first example, we loop over the RDF/item elements and for each item, we print its title.

Reading a RSS 1.0 document (RDF cosher version)

Why is the previous example wrong? Because RSS 1.0 says that the list of items must be found in the RDF/channel/items/Seq/li elements and that fetch the items in RDF/item according to that list.

To facilitate this lookup, we'll create a class for the RDF root element and in that class, define that item elements form a dictionary which key is their attribute "about". With that in hand, we'll loop over RDF/channel/items/Seq/li and and fetch each item:

```
#!/usr/bin/env python

import XmlObject

class RDF(XmlObject.XmlObjectElement):
    dictionaries = {"items": ("item", "about")}

def main():
    feed=XmlObject.XmlObjectDocument()
    XmlObject.XmlObjectElement_RDF = RDF
    feed._ParseUri
("http://xmlfr.org/actualites/breves/breves.rss10")
    print "XMLfr wire:\n"
    for itemUri in feed.RDF.channel.items.Seq.li:
      item = feed.RDF.items[itemUri.resource._value()]
      print ("%s" % item.title._value())
if __name__ == '__main__':
    main()
```

Note that during the definition of the RDF class, we could have added methods and properties to that class to customize its behavior.

Use cases

Data oriented XML

XML data driven classes (not just my implementation) are very well adapted to “data oriented” XML applications where information items must be processed as hierarchies of objects.

New or existing objects

The fact that you can let the API create new classes for you but that you can also provide your own classes means that it is equally useful for new projects where you have no “business classes” to implement the objects that are serialized in XML documents and for projects where you already have these classes.

The fact that Python supports multiple inheritance means that modifying existing classes to be derived from the classes of the API that match the information items isn't an issue like it can be when you're using binding APIs in Java.

Under the scene

4 classes (XmlObject, XmlObjectDocument, XmlObjectElement, XmlObjectAttribute)

This implementation is based on four classes:

- XmlObject is a generic class from which the other classes are derived.
- XmlObjectDocument is the class that matches the notion of a XML document and has additional methods to parse documents.
- XmlObjectElement is the class that matches XML elements. When the application needs to access to an element, a lookup is done to check if a class with a matching name already exists. If not, a new class is dynamically created by derivation from XmlObjectElement.
- Similarly, XmlObjectAttribute is used to create new classes matching attributes.

__getattr__ is overridden to achieve just in time binding

The implementation of the notion of “just in time binding” (ie the fact that classes are created when we need them) is done through overriding the __getattr__ method.

No namespace support (yet?)

There is no namespace support yet (only the local names are used up to now) and I have been surprised to see that it hasn't be an issue for any of the projects I have been using it

so far.

However, I plan to add namespace support as soon as I'll have a project that'll need it!

Difficult to debug

The main thing I need to improve in this API is to make applications using them easier to debug.

The reason why it's hard to debug an application using it is that the exception that is raised when something goes wrong is most of the time an AttributeError.

I find this manageable but that's surely something I'd have to improve if others than just me were using this API.

Where can you get it?

http://cvs.dyomedea.com/cvsweb.cgi/projects/xml-driver/

This is probably not its final home nor even name, but I'll redirect this URL to wherever the project might move.