xml2pm: A Tool for Automatic Creation of Object Definitions Based on XML Instances

Nuno Carvalho[1], Alberto Simões[2], and José João Almeida[1]

[1] Departamento de Informática, Universidade do Minho
{narcarvalho,jj}@di.uminho.pt
[2] Centro de Estudos Humanísticos, Universidade do Minho
ambs@ilch.uminho.pt

Abstract. The eXtensible Mark-up Language (XML) is probably one of the most popular markup languages available today. It is very typical to find all kind of services or programs representing data in this format. This situation is even more common in web development environments or Service Oriented Architectures (SOA), where data flows from one service to another, being consumed and produced by an heterogeneous set of applications, which sole requirement is to understand XML.

This workflow of data represented in XML implies some tasks that applications have to perform if they are required to consume or produce information: the task of parsing an XML document, giving specific semantics to the information parsed, and the task of producing an XML document.

Our main goal is to create object definitions that can analyze an XML document and automatically create an object definition that can be used abstractly by the application. These objects are able to parse the XML document and gather all the data required to mimic all the information present in the document.

This paper introduces **xml2pm**, a simple tool that can inspect the structure of an XML document and create an object definition (a Perl module) that stores the same information present in the orinial document, but as a runtime object. We also introduce a simple case of how this approach allows the creation of applications based on Web Services in an elegant and simple way.

1 Introduction

In todays' distributed world of cloud computing and a multitude of approaches for sharing and distributing resources [1, 8], the need for exchanging information between heterogenous independent systems has become a necessary evil [11, 7].

This interoperability between systems requires information interchange, and to make this information sharing possible and reliable a lot of methods and techniques have been already proven worthy: from the basic RPC (remote procedure call), CORBA (Common Object Request Broker Architecture), or Java RMI (Remote Method Invocation), to the most recent web-oriented approaches, like SOAP web-services (Simple Object Access Protocol) or REST-less (Representational State Transfer) services [5, 12, 13, 6].

Independently of which approach is adopted, a common challenge always ends up being addressed: how to share the information, in a persistent and understandable way.

A sane approach for this problem is to use a structured and well defined document where some basic semantic information about the data can also be included. XML [2] is a mark-up language that has been proven as a good choice to achieve good results on data interchange. This technology is widely available in most development environments (a wide range of tools like parsers, checkers, pretty-printers, and others, are already available), and it exists for quite some time now, which means that it is mature.

Although XML can be an excellent data carrier between heterogenous systems, once the information reaches the application level, processing and handling it in its raw format can be painful and have a deep impact on programming performance (and sometimes, execution performance). Typically it is necessary to transform the data representation maintaining the information content and its semantics.

This transformation commonly requires a task in the parsing family of operations, which will probably be common for every time an application requires to use information stored in XML format. Figure 1 tries to illustrate these common tasks that are performed by applications before they start solving whatever problem they are trying to solve.

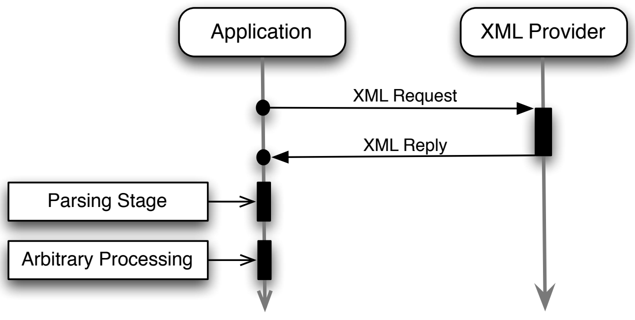

Fig. 1. Typical application workflow prelude.

Notice that this data structure transformation is not required just because XML is a textual format and during runtime programmers prefer to have dynamic structures. Most of the times, the resulting structure will not be used just to make the data usable by the application. It will be the place where data semantics will be analyzed (validating data types, for instance), and where data will evolve through time. Figure 1 also illustrates an *Arbitrary Processing* stage, this is because most of the times another set of tasks needs to be performed after the parsing stage: for example making sure information is standardized, text fields are written in a specific character set, time stamps are all in the same format, etc.

After processing the data, applications want to serialize it again, so it can be sent to other services (or to the service requester). That is, the application needs to convert its data back to XML format, involving yet another usual task on applications that deal with XML formats.

This pattern can be found in many applications, disguised in one implementation or another, but is present most of the times. Since this transformation action is such a common issue why not envisage a systematic and automatic solution to deal with it?

In this article we propose the use of objects, in the context of Object Oriented Programming (OOP) [14], to store the data exchange via XML in runtime, and introduce a tool that can automatically create the required code to create objects that mirror the information and behavior of a XML document. These objects main responsibilities are: to parse the XML data and populate objects proprieties with information; and provide *accessors* and *setters* that allow applications to read and update data.

2 Related Work

There are tools already available that allow a similar approach to the one described in this article. A few of these tools are enumerated here and are very briefly compared to the **xml2pm**. We are not interested in tools that act at runtime, we are only interested in tools that are using during development time to create the required object definitions and parsers. This is mainly because performance issues and because we want the user to be able to add arbitrary tasks to the methods in the object definitions. Some of these tasks can even involve chaging information semantics.

- **autoXML**[10] generates a parser for an XML document given a DTD file. Besides a parser the required structures to mirror the document in memory are also created. This is very similar to the tool introduced in this paper, except for two minor differences: the structures created are not objects *per se* in runtime, and the user is required to manually call the parser.
- **JAXB**[9] is another tool that is able to create classes definitions from schemas that can be instantiated from XML documents.

A more comprehensive and complete list is being maintained by Ronald Bourret [3]. Most of the tools available are either for Java or for C, to integrate this work in another project dealing with ontologies a implementation of these objects in Perl was required, that was one of the major initial motivations for this work.

2.1 Design Goals

This section describes the complete set of design goals that motivated the work described in this paper. Most of the tools referenced in the previous section and their analysis were used as a starting point for devising these, but we feel that none of them by itself could fulfil the entire set.

- Objects definitions are created during application development.
- Objects are able to parse XML documents and mimic the information by themselves, no additional stages are required to be manually done.
- Objects are able to produce an XML document representing their current data.

[3] http://www.rpbourret.com/xml/XMLDataBinding.htm

- Easy to add arbitrary extra processing tasks to information retrieval and update methods.
- Objects can change information semantics more suitable to applications needs.
- Independent objects and parsing capabilities, so that nested structures in a XML document can give origin to their own objects that can be used independently.
- Bigger concern with quickly processing smaller documents than being able to handle huge amounts of data.
- Do not require schemas for the XML documents, this is mainly because most of the times schemas are not available.
- And finally, objects available in the Perl programming language so this tool can be used integrated with the CROSS web portal architecture [4].

3 Prototype Tool

As described earlier the main goal of this tool is to allow the use of objects by applications to reach and manipulate information shared in XML. This means that the new prelude for operations would be something similar to what is illustrated in figure 2.

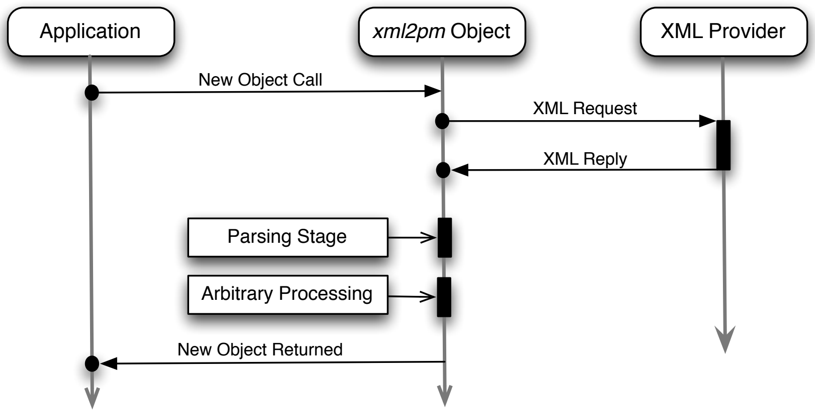

Fig. 2. Application workflow alternative prelude.

In order to do an experimental validation of this approach we developed a prototype tool that can be used to create Perl modules (Perl has no native support for the traditional objects in the context of object oriented programming language, so modules are used to implement them). The tool is named **xml2pm**, in the sense that it processes an XML file and produces one or more Perl modules with the required code to manipulate XML documents with that specific structure.

To use this tool we simply execute it giving the name of an XML file as argument, as shown on figure 3.

[4] http://twiki.di.uminho.pt/twiki/bin/view/Research/CROSS/WebHome

```
$ xml2pm sms.xml
Processing sms.xml ..ok!
Writing Sms.pm .. ok!
```

Fig. 3. Running *xml2pm* on a simple XML file.

In this particular example we processed an XML file that stores the information for a Short Message Service (SMS) message. A file called `Sms.pm` (based on the XML filename `sms.xml`) is created (`pm` is the typical extension for a Perl module file).

This module includes the code to create and use an object, that has the required attributes and methods to provide the information present in any XML document with that same format. This includes all the code required to parse a XML document and populate the object, as well as the code needed to serialize that information again in XML.

```
<?xml version="1.0"?>
<sms>
  <sender>John Lennon</sender>
  <receiver>Paul McCartney</receiver>
  <date>1957-10-02</date>
  <text>
    Hello Paul.
    Do you wanna create a Rock band?
   </text>
</sms>
```

Fig. 4. `sms.xml` file contents.

To illustrate the usage of this tool with a very simple example, consider the XML document shown in figure 4. There are four attributes that are required to describe a message: the sender, the receiver, the date and the body of the message itself. Therefore this is the minimum set of attributes that new objects needs to have. After processing the XML file a Perl module will be generated. Its main structure is shown in figure 5.

Looking up to the code, we can see that the new object type is called `Sms`, it has a `new` method for creating new instances of this object, and a set of setters/accessors.

The constructor (the `new` method) is used for parsing the source XML and setting the objects' attributes with the corresponding values. The tool is able to fetch a file from the filesystem,

```
$sms = Sms->new(file => 'sms.xml');
```

or from an URL (Universal Resource Locator):

```
$sms = Sms->new(url => 'http://randomsms.org/fetch/random');
```

```
package Sms {
  sub new { ... }

  sub sender { ... }
  sub receiver { ... }
  sub date { ... }
  sub text { ... }

  sub to_xml { ... }
}
```

Fig. 5. Perl code to implement the interface to SMS file format.

Setters and getters are created for each attribute. These methods can be called without any arguments, acting as a *getter*, returning the actual value for that attribute. When an argument is passed, they act as *setters*, changing that attribute content.

Finally, a serializing function is also created. Its name is `to_xml` and it returns the object data in XML format. It can also accept some extra parameters, so the XML is written directly to a file.

This object has the same attributes than the original XML document but from the programmer's point of view it is much easier to manipulate in runtime than the XML textual version. The automatic creation of these objects definition is one of the major advantages of using this kind of tools.

Instead of developing a library that, in runtime, analyses the XML file and creates a generic object from it, we preferred to generate static code, that can be used anytime, in the same application or in any other that needs to manage the same kind of data.

This approach has two major advantages: in runtime the information is stored in an object (or a set of objects), instead of an XML document, and the object creation is performed automatically and only once for a specific XML instance. The developer can also easily change the object definition to add extra processing tasks for specific attributes or even change the semantics of the information if required.

Getting back to our previous text message example, we can add an extra action that always make sure that the name value stored in the runtime object is capitalized, as illustrated in figure 6. Now, every time this module is used, this behavior will stick. This ensures flexibility, as we are no longer looking just to a object serialization tool.

4 Case Studies

This section will present two bigger examples:

- The first one is merely a more complex example than the illustrative example from the previous section, where the XML file includes a more complex structure, and there are repeating elements (lists);
- The second one, is a more realistic example, that uses **xml2pm** to quickly create an interface to an XML web service [4].

```
sub sender {
   my $sender = shift;
   $self->{sender} = shift if @_;

   # added transformation
   $self->{sender} =~ s/(\w+)/ucfirst($1)/ge;

   return $self->{sender};
}
```

Fig. 6. Modified *setter/getter* to ensure name capitalization.

4.1 Processing Music Catalogs

In this first example we will illustrate the use of the **xm2pm** tool with a little more complex XML structure than the one shown in section 3. In this example, XML documents are used to represent catalogs of music albums. Each catalog has a name and an associated creation date. The collection consists of a list of albums. For each album we have its title, the name of the artist, the company that edited it and in which year it was released. Figure 7 illustrates an example XML document that represents a music catalog.

```
<?xml version="1.0"?>
<catalog>
  <name>The 80s Collection</name>
  <created>1980-12-31</created>
  <collection>
    <item>
      <title>Empire Burlesque</title>
      <artist gender="male">Bob Dylan</artist>
      <company>Columbia</company>
      <year>1985</year>
    </item>
    <item>
      <title>Like A Virgin</title>
      <artist gender="female">Madonna</artist>
      <company country="USA">Warner Bros</company>
      <year>1984</year>
    </item>
  </collection>
</catalog>
```

Fig. 7. `catalog.xml` file contents.

The first step is to call our **xml2pm** application, feeding in the XML sample document, as illustrated in figure 8. Note that in this case the tool did not create just a module, but a pair of modules. This is **xml2pm** behavior when the XML document has lists.

```
$ xml2pm catalog.xml
Processing catalog.xml ..ok!
Writing Catalog.pm .. ok!
Writing Item.pm .. ok!
```

Fig. 8. Creating a named module from a Catalog XML document sample.

The tool created two modules. `Catalog.pm` implements the main XML object. But when it gets to its `collection` element, a list of items is present. Each one of those items can be seen as small XML document (which root element is `item`), and that is why the tool creates a `Item.pm` module as well. It will handle the data for each item, and `Catalog.pm` element `collection` will deal with lists of this kind of objects. Figure 9 illustrates the `Catalog.pm` file.

```perl
package Catalog {
  use Item;

  sub new {
    ...

      if ($name eq 'item') {
        my $item = Item->new($field);

        push @{$self->{catalog}}, $item;
      }
  }

  sub name       { ... }
  sub created    { ... }
  sub collection { ... }

  sub to_xml     { ... }
}
```

Fig. 9. Perl code to implement the interface to Catalog.

A closer look at the `new` function, which is responsible for creating a new object that represents a catalog, shows the need to handle the collection, nested list of items.

Figure 10 shows the code to handle `item` elements. This example includes an example of how attributes are being handled at the moment (see the `artist` method).

The `new` function is responsible for populating the initial instance and the object, and attributes are later available in a finite function key name to value, which is also stored, besides the value of the element.

```
package Item {
  sub new { ...  }

  sub title { ... }
  sub artist {
    my $self = shift;
    my ($value,$attributes) = @_ if @_;

    $self->{artist}->{value} = $value;
    $self->{artist}->{attributes} = $attributes;

    return $self->{artist};
  }
  sub company { ... }
  sub year { ... }

  sub to_xml { ... }
}
```

Fig. 10. Perl code to implement the interface to Item in a collection.

In this specific case, the `artist` method works differently than the previous examples. The setter do not receive just the value to be set, but also a reference to an associative array (hash reference), with the element's attributes. When used as an accessor, the return value is, also, an associative array, with the key `value` being used for the element's content, and the key `attributes` to the associative array of attributes. We are not very happy with this approach for handling arguments, because sometimes it can result in some confusing code in the applications using the module. We are currently working on other approaches to handle attributes without changing the default accessor/setters Application Programming Interface (API).

Using these modules we can now create simple programs that process information from these kind of catalogs, and produce different results. As a simple example, consider the code in figure 11 an application example to present the catalog in HTML format. This code in elegant and simple, easy to read and maintain.

Now that we have a better idea of what this tool can do, the next section introduces a more practical application example, to clearly show how the use of **xmp2pm** can save time, and increase the elegancy and maintainability of implemented applications.

4.2 Quick Generation of Web Services Clients

A Web Service is a very common case where the XML format is used to transport data. In this section we will show in a set of simple steps how to take advantage of the **xml2pm** tool to implement an application that relies on the information provided by a web service.

We will use an Web Service that, given a city (and respective country) returns, among other information, its geographical position (latitude and longitude).

```
use CGI;
use Catalog;

my $catalog = Catalog->new({file=>'catalog.xml'});

print header, title($catalog->name), "<ul>";
foreach my $item @{$catalog->collection} {
  print "<li> Artist: ".$item->artist->{value},
    "Gender: ".$item->artist->{attributes}->{gender}."</li>";
}
print "</ul>", end_html;

}
```

Fig. 11. Perl code to create a webpage for a catalog.

To bootstrap, we need a sample XML file as returned by the web service. There are different ways to query the web service and store the resulting XML file. One option would be the use of the *curl* command available on most *Linux/Unix* operating systems. This can be accomplished as follows:

```
curl http://.../GeoLookupXML/index.xml?query=braga > location.xml
```

Having a sample of the XML that the web service provides we can build our code using **xml2pm**. We can also use a switch on the command to give it a proper name, as shown in figure 12.

```
$ xml2pm -n Geo::Location location.xml
Processing location.xml ..ok!
Writing Geo::Location .. ok!
```

Fig. 12. Creating a named module from a sample web-service response file.

A new module named `Geo::Location`, that implements objects with that same name, can now be used to query that specific web service in a clean object-oriented fashion. Figure 13 shows a simple application that prints latitude and longitude for a city which name was passed as parameter in the command line (in this case, we are forcing the city to be searched in Portugal). Figure 14 shows the application being executed.

5 Conclusion

In this article we introduced a tool that, by inspecting an XML file, specifies a set of object definitions in Perl, that can be used to represent the same data structure and semantics that are present in the original document. This set of objects can be later used in applications to reach and manipulate the data gathered or received via XML.

```
use Geo::Location;

my $place = shift;
my $url = "http://api.wunderground.com/auto/wui/geo/"
        . "GeoLookupXML/index.xml?query=$place, Portugal";
my $loc = Location->new({url => $url});
print "LAT ".$loc->lat." LON ".$loc->lon."\n";
```

Fig. 13. Code to query the geographic location web-service and print a city latitude and longitude using the object interface as created by **xml2pm**.

```
$ perl latlon.pl braga
LAT 41.58666611 LON -8.45666695

$ perl latlon.pl porto
LAT 41.22999954 LON -8.68000031

$ perl latlon.pl 'vila do conde'
LAT 41.34999847 LON -8.75000000
```

Fig. 14. Output from the geographic location web-service client.

The object itself is able of mirroring the information contained in the XML in his own attributes. The object definition by itself also provides an additional layer that can be used to perform common tasks related to data transformation.

One major drawback of this type of approach are the memory issues that can make this code unusable for big documents, as we are creating a tree of objects mostly like a Document Object Model parser would create. The main difference is that our approach creates code that is specifically created for this type of document.

Also, the fact that this code is generated during programming time means that the programmer can change the behavior of some of the methods, in order to obtain some specific validations or data conversions. Therefore, the created code can be more versatile than a simple generic API to convert XML to and from Objects.

This is a common and systematic approach when implementing applications that deal with data stored in XML format, therefore the use of an automatic tool that can perform most of the work for us with the least information available up front. It has proven useful and allows the implementation of applications much faster and in a elegant and modular way, since all the XML related code is delegated to the object itself. We have demonstrated this situation by showing how to implement a couple of simple example applications described earlier.

6 Future Work

This work is still under heavy development. Therefore, we have a big pile of features we would like to implement. The more imperious tasks that need to be addressed are:

– Currently the object definition is created only by inspecting an instance of an XML source file. The use of Document Type Definition (DTD) documents, or XML Schema documents as bootstrapping source could result in more generic code (we are not dealing with a specific instance) and we can gather some extra information that can be coded (for instance, checking values in accessors methods).
– Some recursive structures will imply the creation of several objects nested in each other. In the current version we are not fully addressing this issue. This situation needs to be well defined to allow the use of this tool in more complex case studies.
– A more natural approach for handling elements' attributes without changing the default accessor/setter behavior could also improve the overall quality of the generated code.

Acknowledgments

This work was partly supported by project CROSS (PTDC/EIA-CCO/108995/2008), funded by the Portuguese Foundation for Science and Technology.

References

1. M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, et al. Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.
2. T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0, 2000.
3. N. Carvalho. OML - Ontology Manipulation Language. Master's thesis, University of Minho, 2008.
4. E. Cerami and S.S. Laurent. *Web services essentials*. O'Reilly & Associates, Inc., 2002.
5. OMG Corba. *Common Object Request Broker Architecture*, volume 2. Revision, 1995.
6. R.L. Costello. Building web services the rest way. *UR L: http://www. xfront. com/REST-Web-Services. html. Ultima Consulta*, 11:2007, 2007.
7. K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. Foster. Grid information services for distributed resource sharing. In *hpdc*, page 0181. Published by the IEEE Computer Society, 2001.
8. T. Erl. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall, 2005.
9. J. Fialli and S. Vajjhala. The Java^{TM} Architecture for XML Binding (JAXB). *JSR Specification*, 2003.
10. J. Kent and H. Brumbaugh. autoSQL and autoXML: code generators from the genome project. *Linux Journal*, 2002(99):1, 2002.
11. T.W. Malone, K.R. Grant, F.A. Turbak, S.A. Brobst, and M.D. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5):390–402, 1987.
12. E. Pitt and K. McNiff. *Java. rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.
13. B. Suda. SOAP Web Services. *Retrieved June*, 29:2010, 2003.
14. RP Ten Dyke and JC Kunz. Object-oriented programming. *IBM Systems Journal*, 28(3):465–478, 1989.