THE PATTERN LANGUAGE OF TRANSLATION

THE INDIRECT APPROACH

ABSTRACT

In this paper, I apply the theory of integration and the indirect approach to translating legacy data. I present a simple means of classifying the complexity of different types of legacy data. I review basic patterns of translation and evolve them towards more complex and powerful models.

THE PATTERN LANGUAGE OF TRANSLATION

THE INDIRECT APPROACH

SIMPLEST PATTERN

The indirect approach lends itself to a pattern of translation that is easily expressed in software. Our objective is to translate existing or legacy data into XML format, that is, syntax. We want the translation to be automatic or at least automated. A program called a parser makes the translation. In our diagram below we show some data D passing through a parser P to create the resulting XML data X.

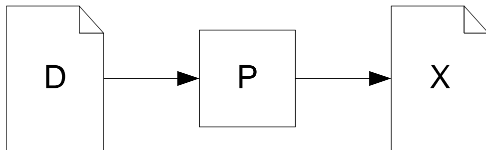

Figure 1 Simplest pattern of translation

We indicate this data in our diagram as a document. Remember that XML is a tree structured data format. For any amount of data in XML format, large or small, there is a root element. The root encapsulates its children. Add to this fact that XML is Internet-centric, and the Internet we think of exchanging files or documents. Thus, we generally think of a collect of XML data as a document.

We could write a specific application P for a specific instance of data D to create the resulting XML document X. Sometimes in system administration or in prototyping we have to do such custom one-of work. The Perl programming language exists and is so popular just because these situations are so common. However, when dealing with the prospect of electronic commerce and trans-enterprise integration, such an approach is impractical. We saw from our discussions of the theory of integration that such an approach cannot be supported on any meaningful scale.

EXAMPLE

Suppose we have a file containing a list of names and phone numbers. The file is comma delimited, that is, commas separate the elements and line returns separate the records.

```
Ricker, Jeffrey, M, 845.555.5465
Lyman, Paul, A, 610.555.2342
Bailey, Michael, E, 703.555.7898
Schoeb, Timothy, J, 703.555.6296
```

We can create a program that translates this particular file.

```
#!/usr/bin/perl

open (IN, "<listofnames.txt") || die "Could not find input file";
open (OUT, ">listofnames.xml") || die "Could not create output file";
println OUT "<?xml version=\"1.0\"?>\n<contacts>\n";
while (<IN>) {
```

```
    ($lastname,$firstname,$mi,$phone) = split($_,/,/);
    print OUT "<contact>";
    print OUT "<lastname>$lastname</lastname>";
    print OUT '<firstname>$firstname</firstname>";
    print OUT "<mi>$mi</mi>";
    print OUT '<phone>$phone</phone>";
    print OUT "</contact>/n";
    }
println OUT "</contacts>";
close IN;
close OUT;
```

The resulting XML document is shown below.

```
<?xml version="1.0"?>
<contacts>
  <contact>
    <lastname>Ricker</lastname>
    <firstname>Jeffrey</firstname>
    <mi>M</mi>
    <phone>845.555.5465</phone>
    </contact>
  <contact>
    <lastname>Lyman</lastname>
    <firstname>Paul</firstname>
    <mi>A</mi>
    <phone>610.555.2342</phone>
    </contact>
  <contact>
    <lastname>Bailey</lastname>
    <firstname>Michael</firstname>
    <mi>E</mi>
    <phone>703.555.7898</phone>
    </contact>
  <contact>
    <lastname>Schoeb</lastname>
    <firstname>Timothy</firstname>
    <mi>J</mi>
    <phone>703.555.6296</phone>
    </contact>
  </contacts>
```

FIRST USEFUL PATTERN

A Perl programmer has to go in and change the parser for every instance of a data file. If the file appears with first name first, for instance, the program must change. In reality, we consider a Perl programmer a scripter, but to keep the example simple we will play along.

What is more useful is a parser P that can translate some particular set of instances of data. Parsers can become complicated rather quickly. As we saw earlier in our discussion of the microeconomics of electronic commerce, the cost of a system lies in the number and expertise of the people required. It would be much more useful if a scripter could make adjustments to the parser to handle the particulars of the instance of data being translated.

Thus we have a set of instructions DD that the parser P uses to translate data D into an XML document X. We will call this set of instructions a data dictionary.

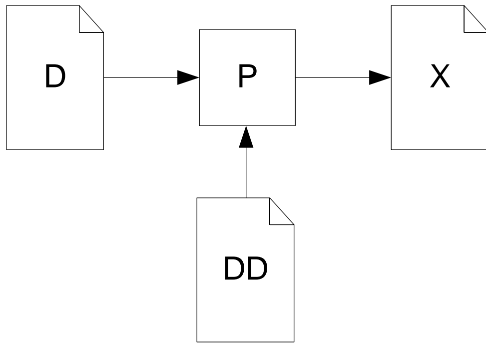

Figure 2 First step toward a useful means of translation

EXAMPLE

We create a file for the field names that is comma-delimited file. One would type the following line in any simple text editor and save it as a separate file.

```
contacts
contact
lastname, firstname, mi, phone
```

For this example, we would want that file to have the name “datadict.txt”. The following Perl program uses the instruction file to parse the data into XML.

```
#!/usr/bin/perl
# open data dictionary
open (INST, "<datadict.txt") || die "Could not open data dictionary";
# load metadata
$rootname = chop(<INST>);
$recordname= chop(<INST>);
%fieldname = split (/,/,<INST>);
$fields = length(%fieldname);
close INST;
# load input file
open (IN, "<listofnames.txt") || die "Could not open input file";
# create output file
open (OUT, ">listofnames.xml") || die "Could not create output file";
# begin writing output
print OUT "<?xml version=\"1.0\"?>\n";
print OUT "<$rootname>\n";
while (<IN>) {
   %line = split($_,/,/);
   print OUT "<$recordname>";
   for ($i=0; $i++; $i<$fields) {
      print  "<$fieldname[$i]>$line[$i]</$fieldname[$i]>";
      }
   print OUT "</$recordname>/n";
   }
# don't forget the closing tags
print OUT "</$rootname>\n";
close IN;
close OUT;
```

The resulting XML file is the same. Now anyone can modify the parser to translate a particular comma delimited file without having to understand Perl.

It would not be too terribly difficult to make the data dictionary itself an XML document. The next chapter walks us through a translation example that uses an XML data dictionary.

CLASSES OF COMPLEXITY

Before we proceed in to more complex patterns of translation, we must first address the different degrees of complexity in existing of legacy data. Not all metadata is created equal. We have found it useful in practice to classify legacy data in three classes. There is nothing terribly scientific about this classification. It simply provides a means to discuss types of data in general terms with other programmers and architects without having to get bogged down in the specifics of the instance.

Class	Description	Example
1	Homogeneous records	CSV
2	Heterogeneous records with published metadata	EDI
3	Heterogeneous records with proprietary metadata	IMS

Table 1 Levels of complexity in legacy data formats

CLASS 1

The first class of data is homogeneous record types. We say homogenous record type because each record has the same structure. Usually, each line represents a record and each item in the list represents a field. The list of names we are using in the example is a class 1.

The most common class 1 data are delimited text files, fixed width files and relational databases (ODBC). Delimited simply means that there is a character or characters that separate fields and records. The most common are comma-delimited (CSV) and tab delimited. Fixed width files determine the separation between records and fields by counting characters. If the value in a field is shorter than the field length, the remaining characters are filled in with spaces.

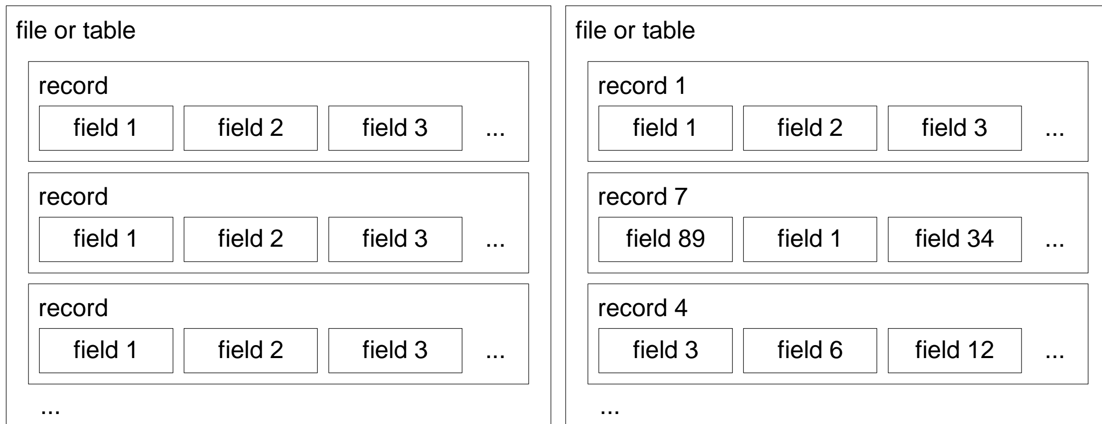

Figure 3 Homogenous record type and a heterogeneous record type compared

Relational databases are also class 1 data types. A relational database is made up of tables which must have homogenous record types. Any query result using standard query language (SQL) will also return a homogenous set of records.

CLASS 2

The second class of data is heterogeneous records with published metadata. In these cases, each line does not have the same structure. We have record types and field types. A record type is a particular set or sequence of field types. A field type might be very generic such as string or integer. Usually, however, field types are more specific with names such as [examples from EDI]. With heterogeneous data, the record invariably begins with a record type identifier, numbers or characters that identify the record type. This allows a parser to know what the field types are of the record. Like class 1, class 2 can be either delimited or fixed width.

The following data file is an ANSI X12 EDI purchase order. ANSI is the American National Standards Institute and X12 is the US standard for electronic data interchange (EDI).

```
ISA*00*       *00*   *08*61112500TST     *01*DEMO WU000003...
GS*PO*6111250011*WU000003 *970911*1039*9784*X*003020
ST*850*397822
BEG*00*RE*1234**980208
REF*AH*M109
REF*DP*641
REF*IA*000100685
DTM*010*970918
N1*BY*92*1287
N1*ST*92*87447
N1*ZZ*992*1287
PO1*1*1*EA*13.33**CB*80211*IZ*364*UP*718379271641
PO1*1*2*EA*13.33**CB*80211*IZ*382*UP*718379271573
PO1*1*3*EA*13.33**CB*80213*IZ*320*UP*718379271497
PO1*1*4*EA*13.33**CB*80215*IZ*360*UP*718379271848
PO1*1*5*EA*13.33**CB*80215*IZ*364*UP*718379271005
CTT*25
SE*36*397822
GE*1*9784
IEA*1*000009561
```

Listing 1 An ANSI X12 purchase order

In this example, line returns separate records and the asterisk (*) separates fields. As you can see, some records are very different than the others. The characters before the first asterisk of each line are the record type identifiers. For instance, REF is a reference record, which has two fields, and a P01 is a purchase order line item record, which has 11 fields.

Up until the advent of XML, electronic commerce was based on class 2 data. It is what we commonly refer to as data standards. Some data standards the author has contented with are listed below.

Standard	Description or purpose	Website
X12	US EDI standard. Encompasses many disciplines.	www.disa.org
EDIFACT	UN EDI standard	
HL7	Healthcare	
AL3	Insurance	
SWIFT	European inter-bank standard	
FIX	US inter-bank standard	
VDA	German automotive industry	
IATA	Airline and travel reservations	

CLASS 3

The difference between class 2 and class 3 is that the metadata for class 2 is published. This difference is very important, especially for XAI. If the data type is shared between many companies, then it is probably published. If the metadata is published, then it will probably change only periodically. Most importantly, if it is published, then a translator might already exist. If one does not exist, then data dictionary can be created from the published metadata. We will discuss this further when we talk about more advanced patterns of translation.

If the metadata is not published, then the opposite is true. The metadata is potentially subject to frequent and irregular changes. It is highly unlikely that an adapter or translator exists for the data, and building one will be a manual process. In other words, the distinction between class 2 and class 3 exists because they indicate different levels of effort in translation and thus integration.

You may come across legacy data that does not fit any of these classes. As we said, it is not terribly scientific. However, the lion's share of the issues you will face will fall into one of these three classifications. It will help you to determine the translation capabilities you will need to integrate your systems. That is, after all, the objective of all this writing.

STANDARD PATTERN OF TRANSLATION

I would like to return now to our example translation. In the previous pattern, we manually created the data dictionary. That's fine with very simple files like the one in our example, but when dealing with complex data, the manual approach becomes cumbersome. For instance, X12 version 4010 has NNN transaction types, NNN record types and NNN field types. Manually creating a data dictionary would be very expensive.

Most data we come across has its metadata published and available. Delimited files often store their metadata in the first record. Relational databases have their metadata accessible as well.

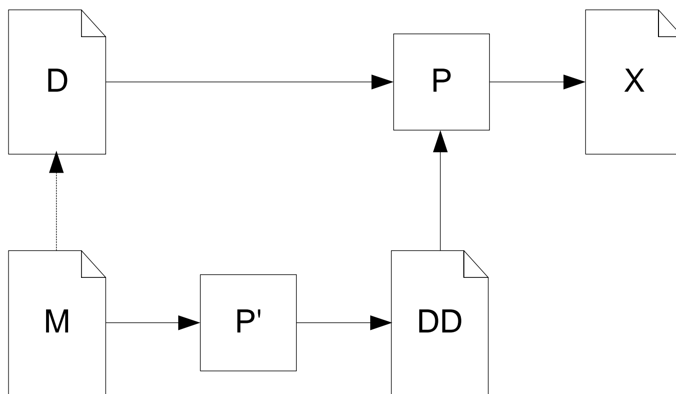

Figure 4 Pattern for translating complex syntax

To improve our pattern, we introduce another parser to automatically create the data dictionary (DD) from the existing metadata (M). This parser we call *parser prime* (P').

Suppose that our files had the metadata in the first row as shown in the following listing.

```
lastname, firstname, mi, phone
```

```
Ricker, Jeffrey, M, 845.555.5465
Lyman, Paul, A, 610.555.2342
Bailey, Michael, E, 703.555.7898
Schoeb, Timothy, J, 703.555.6296
```

Our parser prime would simply chop the first line and store it in a separate file. Our parser would need to change to ignore the first record of the file.

With this simple example, the parser prime seems a bit trivial. When dealing with something like large relational databases, EDI or even SAP IDOC, however, the parser prime is indispensable.

EXTENDED PATTERN

You might have noticed that our simple Perl parser can only handle a particular set of files. These files have homogenous record type, with comma-delimited fields and line returns between records. Our parser can handle any file that matches this description. This description we call the *metamodel*.

The metamodel (MM) is an abstraction of the metadata (M). It confines the behavior of the parser prime (P').

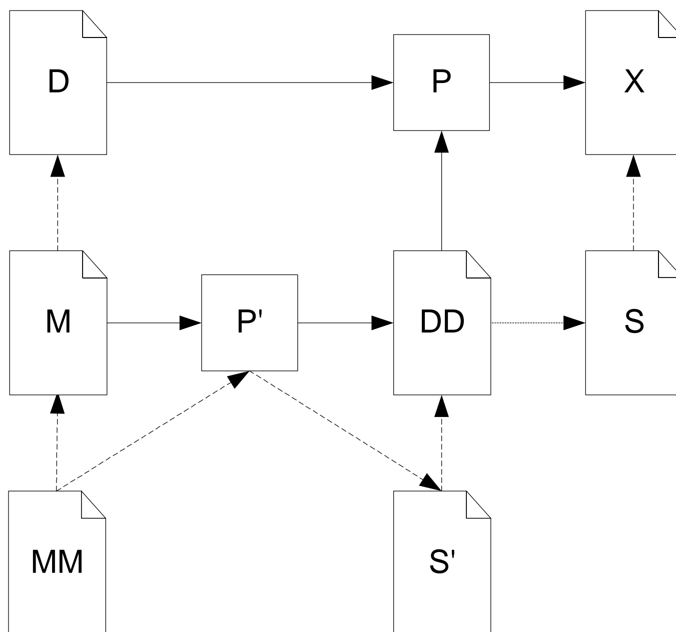

Figure 5 An extended pattern for enterprise level translation

Another consideration for extending this pattern centers on the data dictionary (DD). The parser is translating non XML files into XML. The resulting XML should have a schema (S). The data dictionary would define particulars of this schema. Finally, the data dictionaries themselves are XML documents that have a schema (S') that is different than the schema (S) of the translated documents.

As we pursue this line of thought, the complexities begin to collapse in upon themselves an we gain a higher, more elegant pattern language for translation. What makes this higher pattern possible is the advent of XML Schema, an XML language that can describe XML documents. XML Schema allows us to create a self-defining system, made most evident by the XML Schema document that defines XML Schema. (It's enough to make Kurt Gödel's brain work for a minute.)

To create our advanced pattern, we extend the XML Schema language to include the particulars of a text-based format, most specifically the delimiters. This extension is relatively trivial. It is, after all, extensible markup language. Once we make such an extension, the same schema defines both the target and the source. With this sort of translator in place, there is no differentiation between non-XML data and its XML equivalent.

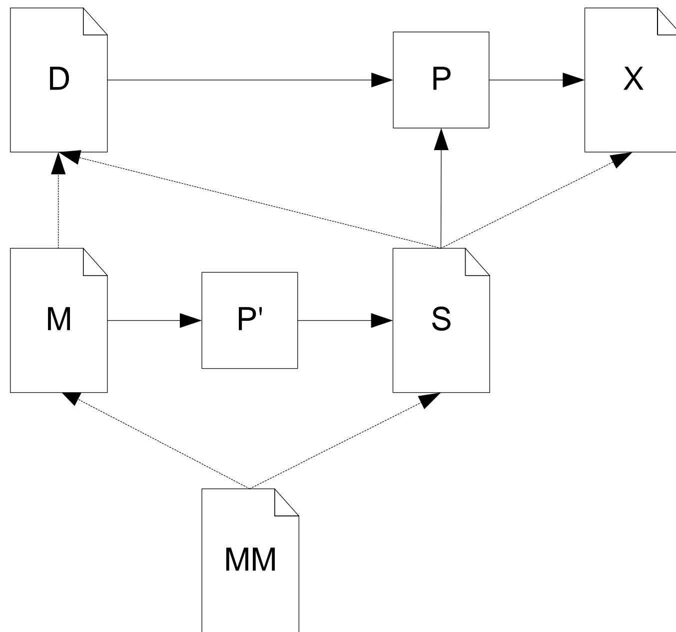

Figure 6 Advanced pattern for automated translation

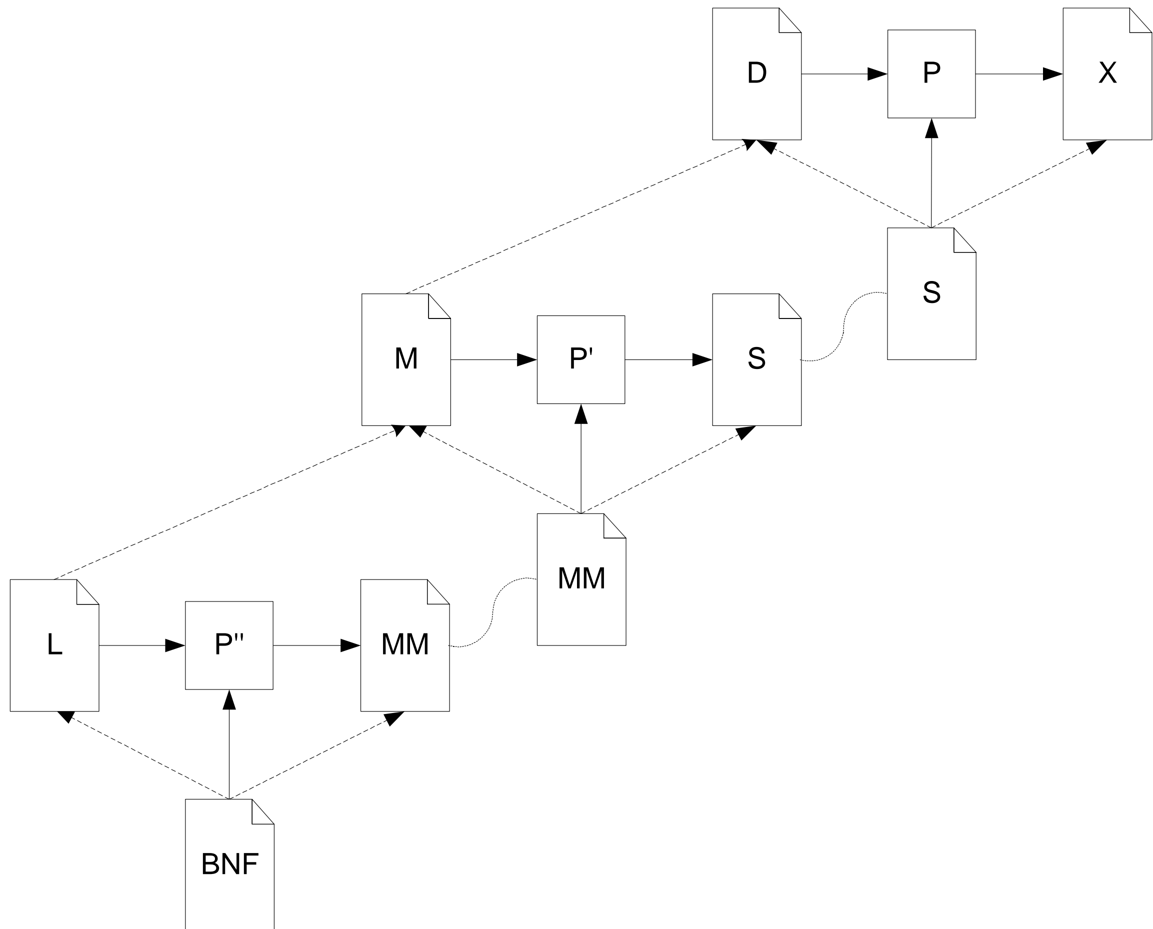

Figure 7 Conceptual, recursive, universal model of translation

The same holds true for the parser prime (P'). Conceivably, the parser and the parser prime could become the same thing. In fact, it is further conceivable that a parser double prime (P") could exist that takes in a definition of any arbitrary data structure to produce the metamodel based on the source language (L) itself. Perhaps such a definition of arbitrary data exists with Backus Naur Form (BNF) used by the W3. The result would be a recursive, universal translator.

There are, of course, some practical limitations to implementing such a translator such as accessing the metadata. For instance, accessing the metadata of a relational database is very different than an EDI data dictionary. Nevertheless, as standards progress and consensus builds, more of this universal translation will become practical.