Printing International Text Using Java, XML, XSLT, and PDF Forms: A Case Study

Geoff Adams
FGM Inc., Dulles, VA USA
gadams+iuc@fgm.com

ABSTRACT

Designing a single software system to generate documents in diverse locales presents interesting challenges. Legal documents are especially difficult, because they must meet strict requirements for format, graphics, and typography.

This paper describes the design and implementation of a system that takes multilingual XML data from a Java program and generates localized printed documents. The document formats are customizable to meet the stringent formatting and typographical requirements of numerous end user sites, each of which requires customized legal documents in its own language.

To implement this system, we chose to use the PDF forms feature of Adobe Acrobat. We transform the XML data supplied by the Java program into PDF Forms Data Format using XML Stylesheet Transformation (XSLT), and send the result to Acrobat for printing. This paper describes the advantages of using this approach, as well as the problems we have encountered along the way, and how we worked around them. In particular, Acrobat's support for Unicode text and various fonts that support all the necessary glyphs has improved dramatically over the past several years.

1. Introduction

1.1. The system

We have created a software system that collects information relating to official forms. The primary goals of the system are to help users organize and analyze the data entered into the system and to facilitate printing the data onto the forms.

As an example, an office might be responsible for issuing licenses for some activity, such as driving a car or opening a restaurant. The issuing office would want to collect information about the license applicant, including name and address, as well as information about the activity to be licensed. For some activities, such as driving, background information such as the applicant's driving record would be desired as well. When the decision has been made to issue the license, the office must produce the official document – the license.

Our system facilitates this process by collecting the relevant information in form-like windows on the computer screen and storing the data in a database. The system then organizes and presents relevant data, including the just-entered information as well as other stored data that helps the office make decisions based on the information. Finally, the software generates and prints the completed form.

1.2. The desired printed output

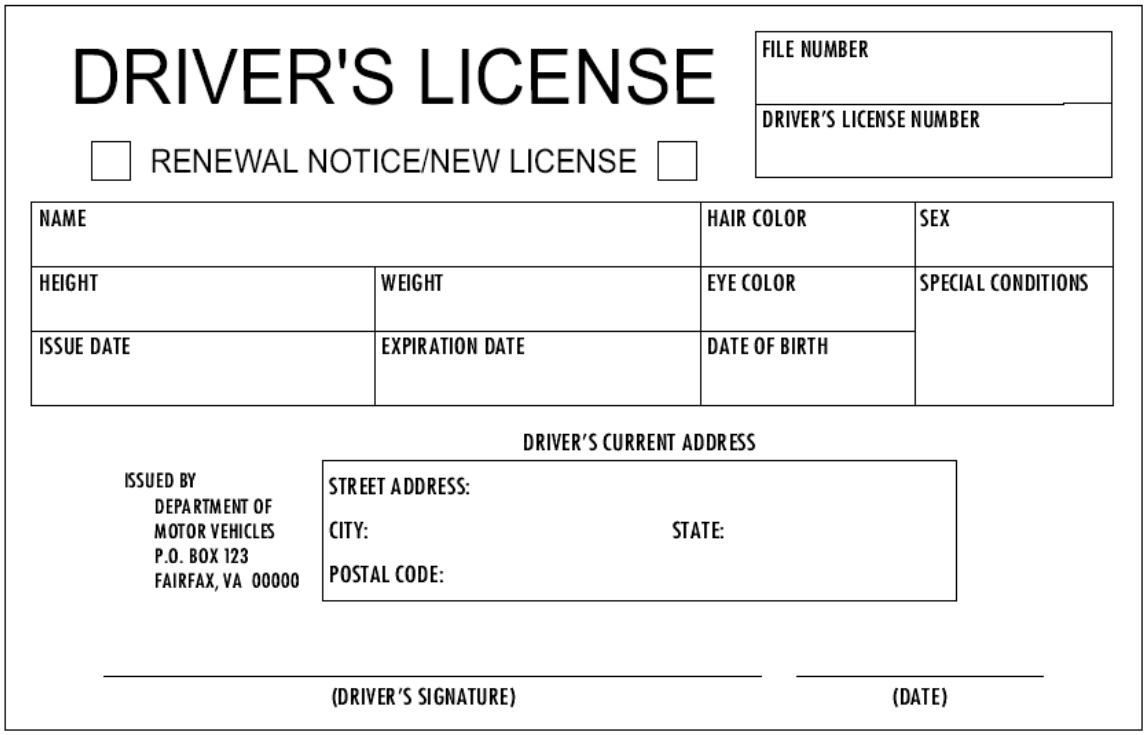

Figure 1. An example official form

1.2.1. Legal requirements

In many cases, the printed form is a legal document. Therefore, it must meet legal requirements that can be quite strict. The common requirements of such printed forms include document size, the position of lines, boxes, and emblems, and the exact wording, appearance, and location of text. In some cases, the document must be printed on special paper that has a watermark or a multi-colored background pattern.

To meet all of these requirements, our software must provide significant and fine-grained control over the way the documents are generated. In some cases, for instance, if a box on the form is a few millimeters too far from the edge of the paper, the printed output is useless.

Note that our software is designed to print the entire form onto the appropriate paper. We are not simply filling in the blanks on a pre-printed form. In most cases, we are printing the form itself. This approach greatly simplifies the workflow at offices in which our software is installed.

1.2.2. Flexibility

The software is intended to operate in offices in any country in the world. This intended use requires the system to handle diverse languages as well as a wide variety of printed documents.

It would be most desirable for a single implementation of the software system to be capable of printing any of these forms in the appropriate languages, simply by providing configuration information at each site.

1.3. Implementation

We have chosen to implement this system using Java. This choice gives us the advantage of a cross-platform system without doing any explicit work to make it so. Even more important, Java's built-in support for Unicode allows us to operate seamlessly with many languages.

Because our system is implemented entirely in Java, we have encountered very few problems related to language support. We have localized our graphical user interface for a number of languages, and Java handles the multilingual text input for us. (This is not to say that Java implementations have been flawless. Indeed, we have encountered some platform-dependent difficulties with support for text input in various languages.) Java also handles the display of multilingual text on the computer screen.

Very little effort was required to create a system that can interact with the user in his or her preferred language. And, because Java's native text encoding is Unicode, storing the information is relatively simple. However, printing the multilingual text – while maintaining tight control over the exact presentation of that text – has proven difficult.

This paper describes the path we chose in implementing a solution, and provides example documents and relevant source code implementing our solution.

2. Initial planning

We had implemented a previous version of the system that used the printing support in Java to print documents directly to a printer. Aside from the considerable bugs in this implementation, direct programmatic printing did not produce documents that could be used by any legal process. The primary reason for this was that it was infeasible to produce the exact output required by each individual site – doing that would have required writing Java code to print each possible legal document. So, rather than attempt to print any particular valid legal document, our first implementation could produce only a single printed format that included all of the collected data in an unattractive table. While this provided an interesting proof-of-concept for the rest of the system, it meant the system could not be used in a production environment.

It was time to revise the document printing system. The experience we had with that first printing implementation guided us in designing the new printing system.

2.1. We don't want to deal with printing directly (or, Printing is difficult)

One of the primary lessons we learned about printing from the previous implementation was that direct printing is to be avoided. There were a considerable number of bugs in either the printing support in Java or our use of it, some of which caused the program to hang during printing. Furthermore, printing has always been one of the areas that differ greatly from platform to platform. These differences meant that we had to debug and sometimes write separate code for each platform we chose to support. This approach did not fit well with the cross-platform goal of our project.

It seemed clear that we wanted to produce printable documents in some document format that is well-specified and well-supported across a wide range of platforms. Then, we could use built-in operating system support or available third-party support to handle the printing of these documents.

It was vitally important, however, that the document format we chose be capable of printing text in a wide range of languages.

2.2. A first candidate output format: PostScript

Our initial thought was that we wanted to produce PostScript output. PostScript is a page description language. It is designed to give the entity creating the PostScript document complete control of the content and layout of a printed page. This solution seemed perfect. We could produce PostScript that described an arbitrary legal document, and then simply send it off to the operating system to be printed.

In reality, it is not that simple. First, producing arbitrary PostScript code may not be straightforward. While the language is very flexible, we would still need to produce the appropriate PostScript code for each individual legal document for each site. To produce this PostScript code, we would be writing separate, rather complex document-printing Java code for each of these many document formats. Installations of the system at a new site, with new legal document formats, would require modifying the Java code. This requirement would preclude delivering the system as a generic software program, and it would prohibit the sites from modifying the set of legal documents later.

To solve these problems, we could add another layer of abstraction. Instead of producing the PostScript code directly, we could create output in some intermediate format, and use a converter to convert that format into PostScript. Ideally, our Java program would produce a document that contained just the relevant data without any presentation information. We would then use some sort of a layout description document that would instruct the converter where to place the data on the page.

There is one final problem with the PostScript approach, however. Not all systems are capable of printing PostScript documents. For instance, some printers use other page description languages, such as PCL, and some accept only raster data. A system, such as

Windows, that is unable to convert PostScript into raster data will not be able to print PostScript documents to such a printer.

It seemed desirable to take yet one step farther away from the printer, and produce documents that some cross-platform program would be able to print as appropriate on whatever system it is running on.

2.3. A second candidate: Formatting Objects (XSL-FO)

Our system is capable of exporting its data in the Extensible Markup Language, XML. XML is a convenient, human-readable data format that is supported by an array of Java language tools and packages. XML has a number of other advantages as a format for storing and transmitting pure data, and we can make use of it for our document printing as well.

One way to take advantage of our system's XML capabilities would be to use Formatting Objects. A Formatting Objects document contains a collection of text, graphics, and positioning and layout information. In many ways, Formatting Objects serve a similar purpose to PostScript, the primary difference being that Formatting Objects are expressed in an XML format.

Formatting Objects, also known as XSL-FO, are defined in the Extensible Stylesheet Language (XSL) specification. The XSL specification also defines a transformation process called XSL Transformations (XSLT) that uses an XSL stylesheet to transform an arbitrary XML document type into Formatting Objects. Like Formatting Objects, XSL stylesheets are represented in XML format.

The alphabet soup of acronyms can somewhat obscure the elegance of XSL Transformations. The intent of the specification is that the source XML document contains the data to be presented, and the XSL stylesheet describes the intended presentation (also called style). The result of transforming the XML data using the stylesheet is Formatting Objects, which can be rendered on an output device. This is a close fit for what we are trying to do.

2.3.1. Advantages

There are several theoretical advantages to using XSLT and Formatting Objects. First, our Java program already produces an XML document in a well-defined format. No modifications are needed in order to feed this XML document into the XSLT process.

Second, and more important, the use of a separate stylesheet describing the intended presentation of the information is very convenient. Each site could simply use a different stylesheet (or set of stylesheets) to describe their legal documents. No Java code would be involved in the layout of the legal documents (other than a generic XSLT transformation utility, meaning we could ship a generic software package to any site. We could provide a separate CD containing the stylesheet files needed by each site, and the

software manuals would provide instructions on how to install the stylesheet files into the system.

Finally, the Formatting Objects specification is freely available and well-defined. There should soon be many implementations of software, including cross-platform software, that could render Formatting Objects on the screen as well as print them.

2.3.2. Disadvantages

That was the thought, anyway.

We were looking into this plan right around the time that Microsoft was announcing that the upcoming version 5.0 of their Internet Explorer browser would have XML capabilities. It seemed clear that the browser would read XML documents together with XSL stylesheets describing how to render the information via Formatting Objects. Internet Explorer runs only on a few platforms, but those platforms would cover at least a large portion of our user base. We could use one of the many other Formatting Objects-aware products that would be coming along shortly to complete the picture.

Unfortunately, Internet Explorer's XML support does no such thing. It does have very limited support for using XSLT to render XML documents as HTML, but that technique cannot produce legal documents. Even when combined with Cascading Style Sheets (CSS), HTML is a content markup language, not a page description language. There is no way to produce an HTML document that will meet arbitrary legal requirements for layout.

In the time since then, no other software has become available to fill this niche. There is really only one software package available that can render Formatting Objects. It is known as FOP, and it will be discussed below.

There is another significant disadvantage to this technique: It is not scalable. For each legal document to be printed, an XSLT stylesheet must be written to convert the XML document format into Formatting Objects. Writing this stylesheet is a complex process requiring a programmer. Aside from this challenge, writing Formatting Objects code by hand to draw lines in exactly the right place and choosing the exact typeface and positioning for each line of text would be very tedious. Handling graphics, such as seals and emblems, would be another problem.

Clearly, transforming the XML data into Formatting Objects is, at best, an incomplete solution.

2.4. A third candidate: Formatting Objects to PDF (FOP)

As mentioned above, there is a package called FOP that renders Formatting Objects into PDF documents. Adobe's Portable Document Format, PDF, enjoys support on a large number of platforms, including those used at our deployment sites. Furthermore, the PDF

specification is publicly available, which – aside from affording some stability to projects that choose to use it – is really what makes it possible for a project such as FOP to exist.

FOP is an open-source project with a well-understood license, making it appropriate for inclusion in our system. The open-source license also means that, should FOP lack some capability that we require, we could add it and submit the improvement back to the FOP maintainers.

So, we could use XSLT to transform our XML data into Formatting Objects, and then pass the result on to FOP, which would generate a PDF file. Finally, we would hand the PDF file to Acrobat Reader, which would allow the user to preview and then print the fully-rendered document. This design seems like a complete solution.

2.4.1. Advantages

Rendering Formatting Objects into PDF provides the needed flexibility. We would be able to produce any desired document format without software modifications. All we would need to do would be to write an XSLT stylesheet for each desired printed document format.

FOP is also written in Java, and as such will run on all of our target systems. Acrobat Reader exists for Mac OS, Windows, and the most popular Unix operating systems. Furthermore, the open source license terms are attractive.

2.4.2. Disadvantages

Unfortunately, the scalability problem mentioned above in the context of Formatting Objects applies here as well. Creating document layouts for each site would require writing complex stylesheets that transform our XML data into XSL-FO. Writing such stylesheets requires thorough knowledge of our XML format, XSLT, and XSL-FO. To create the printed document layout for any given site, a programmer must determine the XSL FO code to describe that layout, and must then work backward from that desired code, writing an XSLT stylesheet that will generate it from the XML. This task is an involved undertaking for a skilled programmer.

Finally, at the time we were investigating it, FOP was unable to incorporate graphic images into the PDF document. Embedding graphics is a reasonably difficult problem, because it requires converting the graphics from whatever format you want to use into the internal raster stream format of PDF.

We theoretically could have written graphics support for FOP and contributed it back to the project, but it would have taken longer than our schedule allowed. Therefore, using FOP would not solve our problem.

(Note that in the time since our initial evaluation, FOP has gained the ability to incorporate graphics into the PDF documents it produces, and it has improved in many

other ways as well. The scalability problem, however, would still be an impediment in our particular application.)

2.5. Prime candidate output format: PDF Forms

It turns out that we need not abandon the advantages of PDF as a document output format because of difficulties in producing it programmatically. The investigation into PDF led us to examine a feature of Acrobat and Acrobat Reader that has proven to be very useful.

Acrobat can generate and use PDF Forms. The primary intended use of PDF Forms is to provide a user interface for filling in form data. The user is presented a form in Acrobat Reader, for instance, and fills in the desired fields. Finally, the user submits the completed form, and the data are sent to the desired location – perhaps to a web server over the Internet.

The filled-in form data are encoded in a Forms Data Format (FDF) file, a format that is specified in the PDF specification (first appearing in revision 1.2). The FDF file contains a reference to the relevant PDF form file. Because two files are linked this way, Acrobat can read the form data (in the FDF file) and automatically populate the form fields at a later time.

We can take advantage of this feature by generating an FDF file programmatically. All we have to do is instruct Acrobat Reader to open the FDF file, and Acrobat Reader will locate and open the corresponding PDF file and populate its form fields with the data from the FDF file. This approach matches our desired usage pattern very well. The Java program can provide FDF files containing all the relevant data to be inserted into the legal forms, and Acrobat Reader will handle all the details of rendering and printing the result.

Furthermore, almost any tool can be used to create the PDF form document. Creating the form to be filled in is no longer the exclusive domain of the programmer. A graphic artist can use a program such as Adobe Illustrator to create the forms in a straightforward way, and use Acrobat Distiller or even a “print-as-PDF” technique to generate the PDF file. Alternatively, offices that already have their forms in some electronic format can simply use the “print-as-PDF” method to generate the PDF file. Once a PDF file is in hand, Acrobat (not Acrobat Reader) is used to mark the form fields in the PDF file. Creating a form that meets any legal requirements is relatively straightforward when using graphical tools intended for the job.

The last step is to convert the XML data from the Java program into an FDF file. XSLT, described above, can do this with ease. The XML Stylesheet Transformation can produce output documents that are in formats other than XML by specifying “plain text” as the output format. All we need to do is write an XSLT stylesheet that produces a valid FDF file with all the fields we might want to use.

As we’ll see, however, handling international text complicates the issue considerably.

3. First workable solution: Acrobat 4 with font mapping

At the time we first implemented this solution, Acrobat 4 was the current version. Using Acrobat 4 (and later, 4.5), we have successfully printed legal documents in Western European languages, over 10 Central and Eastern European languages, and some Cyrillic-based Asian languages.

However, there is a serious problem with Acrobat 4: Support for text other than US-ASCII is very limited. We had to devise an elaborate work-around for that problem.

3.1. Limited Unicode support

Acrobat, as of version 4, limited the choice of fonts available for use in form fields to a set called the "Base 14 Fonts." These 14 fonts are:

- Courier (Regular, Bold, Oblique, and Bold Oblique)
- Helvetica (Regular, Bold, Oblique, and Bold Oblique)
- Times (Roman, Bold, Italic, and Bold Italic)
- Symbol
- ZapfDingbats

These Base 14 Fonts do not include the characters needed by languages other than those of Western Europe. In fact, the default encoding of text in PDF documents is called "PDFDocEncoding," and it conveniently matches Unicode up through code point 255. This means this encoding can handle only US-ASCII and ISO 8859-1 (Latin 1) text.

Note that the Base 14 Font problem only affects the form fields themselves; the body of the PDF document can use whatever fonts are available on the computer producing the PDF document. However, not being able to print localized text in the form fields is a catastrophic fault.

3.2. Handling text in various languages

One of our basic goals is to print text in forms in a variety of languages. Clearly, we needed to develop a work-around for the Base 14 Font problem. We tried a number of techniques, but we were thwarted at every attempt. Finally, we resorted to hacking Acrobat Reader itself.

The Base 14 Fonts are permitted for use in form fields because they are included with every copy of Acrobat and Acrobat Reader. In fact, they are included within the Acrobat application folder hierarchy, rather than being installed in the operating system's font folder. These fonts are treated specially by Acrobat, and Acrobat is unaccustomed to people interfering with them. This special treatment is why Acrobat allows the use of only these fonts in form fields; these are the only fonts it can guarantee will be available exactly as expected.

Acrobat will ignore additional fonts added to its internal font folder. So, the only way to use a font other than the Base 14 in form fields is to trick Acrobat.

3.2.1. Create a custom font

We trick Acrobat by creating a custom font that contains all the characters required by a site's languages, and then give that font the same name as one of the Base 14 fonts. We then install this font into the appropriate folder within the Acrobat Reader application itself.

We usually choose to replace the Base14 font Courier Bold with the custom font, because Courier Bold is not normally used on forms, and its mono-spaced metrics don't cause problems when characters are mapped into unusual places. The font metrics, or spacing of individual glyphs, matters because Acrobat will use its knowledge of the replaced Base 14 font when drawing the glyphs from the custom font. Consider what would happen with a proportionally-spaced font if glyphs were arbitrarily replaced with glyphs of a different width. It is difficult to achieve a reasonable final output unless you use a mono-spaced font.

To create the custom font, we usually start with a font (preferably also mono-spaced) that the site already uses. We edit the font using a program such as Fontographer or FontLab, changing the PostScript glyph names of the substituted characters to appropriate values. (Acrobat essentially looks up glyphs in a font by PostScript character names, such as "`/daggerdbl`" and "`/Odieresis`". The PostScript names for the characters in PDFDocEncoding from positions 128 through 255, with the exception of 159 and 173, may be used.) We then save this edited font in the appropriate formats for Mac OS, Windows, and Unix.

Finally, we provide the font and instructions describing how to install the font into Acrobat Reader to the sites that need to use them.

(Note that you must be careful to respect the licensing requirements of your source fonts when creating a custom font. Most fonts are protected intellectual property.)

3.2.2. Describe the encoding of the font

The custom font we have created essentially has a custom encoding. Acrobat will believe that the font has the standard encoding, so if we construct the text in the FDF files with the appropriate character values, Acrobat will display the desired glyph.

In order to construct the text strings in the FDF file, we need to convert the strings from Unicode to the new encoding. We do this by first creating an FDF file with Unicode text, and then passing the entire FDF file through a character-by-character encoding conversion. The new encoding is described in another XML configuration file that we provide along with our modified font. This XML file must be installed into the system in the appropriate place before documents can be generated.

3.2.3. Significant problems remain

This technique is ugly, difficult, and just wrong. If modifying Acrobat Reader itself to enable printing documents using custom fonts with non-standard encodings weren't dubious enough, consider that doing this renders the Acrobat Reader unable to display normal documents correctly if the documents happen to use the replaced font. Aside from these intellectual problems, this technique has some serious practical drawbacks.

PDF is closely related to PostScript, and the way the two languages handle fonts is very similar. In fact, the Base 14 fonts are all standard PostScript fonts. So, when Acrobat prints a PDF document, it optimizes the operation by not sending any Base 14 fonts to the printer. This means that any PDF form that uses a modified Base 14 font will be printed using the *original* PostScript font residing in the printer. This results in garbled printed documents.

To work around this problem, the user must use the option "Print as Image" from Acrobat's printing options dialog. This causes Acrobat to render the PDF document into a bitmapped image and then send that image to the printer. Not only is this inefficient and slow, but depending on the operating system used, it may look bad, too. Furthermore, the user must remember to click the "Print as Image" checkbox before printing each document. The technique does mostly work, however.

Another problem is that some forms require data to be added directly to the PDF form immediately prior to printing. For instance, some form fields may contain data that are necessary in the printed output, but that are not available from the Java program generating the documents. Unfortunately, when using a custom Base 14 font, text fields in the PDF form cannot be edited within Acrobat. This inability exists because Acrobat reverts to using an operating system font for on-screen editing, and the operating system uses a different encoding than the custom encoding.

Finally, this font-modification technique is a maintenance issue. Each new site at which the system is to be installed might have a unique set of languages that it needs to use, requiring us to create a new font and font encoding map for each country. Furthermore, some sites may need to use more characters than can fit in the small space between code positions 128 and 255. These sites cannot be accommodated using this technique.

3.3. Moving on

Despite all of the problems and annoyances, we successfully used this implementation in a production environment for a couple of years. However, maintaining this solution was taking up far more time than we would have liked. We were fondly looking forward to the arrival of Acrobat 5, with its promised improved international language support.

4. A better solution: Acrobat 5 with Unicode

Acrobat 5 introduces one new feature that, above all others, makes it a far superior solution for us. It allows the use of fonts other than the Base 14 in form fields. This capability means that you can select off-the-shelf fonts that contain glyphs for the languages of concern for a given form, and you need not modify the installed copy of Acrobat to use them. In fact, in many cases, you can embed the font directly into the PDF document, so the form will be viewable (and printable) on any machine.

Being able to select any appropriate font for the form fields means that we need no longer be concerned with custom font encodings, modifying Acrobat, and printing documents as images. The result is a greatly simplified document generation process. I will describe the process using Acrobat 5 in detail below.

Another PDF feature that was partially supported by Acrobat Reader in the past is now much more useful. Text strings in PDF documents are, by default, encoded using PDFDocEncoding, as mentioned above. However, if the string begins with the two-byte Unicode byte-order marker U+FEFF, the string is interpreted as UTF-16-encoded. This encoding was not much use in Acrobat 4, because none of the Base 14 fonts contained glyphs for code points outside PDFDocEncoding. However, when used with Acrobat 5, together with the appropriate fonts, this feature (theoretically) allows us to generate and print arbitrary human-language text in PDF form fields.

I include the qualifier “theoretically” in the previous paragraph because there are still limitations in Acrobat’s Unicode support. Some Unicode characters will, if present in a string, cause Acrobat to crash. Others will simply display missing or incorrect characters.

To correct some of these problems, the French company WinSoft has adapted Acrobat 5 to improve the support for Unicode text in PDF documents. They have released two versions of Acrobat 5: Acrobat 5 CE and Acrobat 5 ME. These include improved support for Central European languages and Middle Eastern languages, respectively. The main selling point of these enhanced Acrobat products may be the localized user interface they provide, but for our purposes, the bug fixes relating to the handling of Unicode text in FDF strings are the key benefit.

It would be nice if Adobe were to incorporate the Unicode-handling fixes back into the base Acrobat product, because there would then be a single Acrobat Reader that was capable of handling arbitrary Unicode text. In reality, any given site tends only to deal with the languages of their country and their neighbor countries, so one of Adobe’s standard (Western European) Acrobat or WinSoft’s CE or ME version is sufficient.

We have only worked so far with languages using Roman and Cyrillic scripts. Soon, however, we’ll be working with right-to-left scripts and as well as Asian languages such as Japanese, so we will see how Acrobat holds up to these challenges.

4.1. Here's how it works

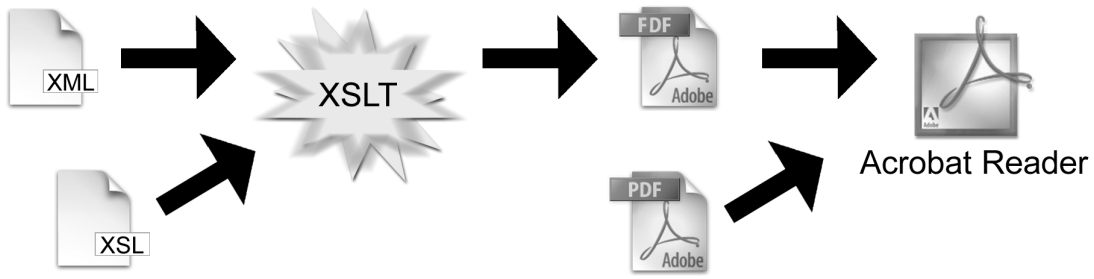

Figure 2. The process of generating a printable document

As mentioned, the Java program generates an XML file as its output. Figure 3 shows an example XML file that might be used with the Driver's License in Figure 1. This XML file is in a particular defined format (known as its schema) that contains all the data relevant to our legal domain. Because we know all the possible data fields that can exist in the XML file, we can construct an FDF file that also contains all the possible data. Converting the XML file to FDF is then simply a format conversion.

```
<?xml version="1.0" encoding="UTF-8"?>

<DriverLicense type="NEW">

    <FileNumber>0183247</FileNumber>
    <LicenseNumber>X987654</LicenseNumber>

    <Driver>
        <ID>6537</ID>
        <FirstName>Иван</FirstName>
        <MiddleName>Иванович</MiddleName>
        <LastName>Иванов</LastName>

        <BirthDate>1977-12-23</BirthDate>
        <Weight>80 Kg</Weight>
        <Height>175 cm</Height>
        <Sex>Муж.</Sex>
        <EyeColor>Коричневые</EyeColor>
        <HairColor>Коричневые</HairColor>

        <Address>
            <StreetAddress>ул. Петрова 1, кв.5</StreetAddress>
            <City>Москва</City>
            <PostalCode>123456</PostalCode>
            <CountryCode>RU</CountryCode>
        </Address>
    </Driver>

    <IssueDate>2002-05-22T14:29:00.128EST</IssueDate>
    <ExpirationDate>2003-05-22T14:29:00.128EST</ExpirationDate>

    <LicenseOfficer>Frank</LicenseOfficer>

</DriverLicense>
```

Figure 3. An example XML file produced by the Java program

Of course, format conversions are rarely as simple as they might first seem. In this instance, there is a significant difference in the structure of the information. The data in

our XML schema are hierarchical, while FDF is a flat format. That is, in our XML schema, there are data elements that contain other data elements. FDF, on the other hand, consists of a simple list of name-value pairs. So, there is some effort involved in designing a system of naming the fields in the FDF document.

In our simplistic example, it is fairly straightforward to create a list of FDF field names that can contain a flattened version of our XML schema. Figure 4 is the listing of an FDF file that contains the information from the XML file in Figure 3. The strings in parentheses after each "/T" are the field names; the strings in parentheses after each "/V" are the corresponding values to be filled into the PDF form.

```
%FDF-1.2
% Hand-generated via XSLT
1 0 obj
<<
/FDF << /F (driver-license.pdf) /Fields 2 0 R >>
>>
endobj
2 0 obj
[
<< /T (fileNumber) /V (0183247) >>
<< /T (licenseNumber) /V (\376\377\004%\0009\0008\0007\0006\0005\0004) >>
<< /T (licenseIssueDate) /V (\376\377\0002\0002\000 \004<\0040\0049\000 \0002\0000\0000\0002) >>
<< /T (licenseExpirationDate) /V (\376\377\0002\0002\000 \004<\0040\0049\000 \0002\0000\0000\0003) >>
<< /T (specialConditions) /V () >>


<< /T (newLicense) /V (X) >>

<< /T (driver_id) /V (6537) >>
<< /T (driver_name) /V (\376\377\004\030\0042\0040\004=\000 \004\030\0042\0040\004=\004>\0042\0048\004G\000 \004\030\0042\0040\004=\004>\0042) >>
<< /T (driver_birthDate) /V (\376\377\0002\0003\000 \0044\0045\004:\000 \0001\0009\0007\0007) >>
<< /T (driver_height) /V (175 cm) >>
<< /T (driver_weight) /V (80 Kg) >>
<< /T (driver_eyeColor) /V (\376\377\004\032\004>\0048\0048\004G\004=\0045\0042\004K\0045) >>
<< /T (driver_hairColor) /V (\376\377\004\032\004>\0048\0048\004G\004=\0045\0042\004K\0045) >>
<< /T (driver_sex) /V (\376\377\004\034\004C\0046\000.) >>


<< /T (driver_address) /V (\376\377\004C\004;\000.\000 \004\037\0045\004B\0048\004>\0042\0040\000 \0001\000,\000 \004:\0042\000.\0005) >>
<< /T (driver_city) /V (\376\377\004\034\004>\004A\004:\0042\0040) >>
<< /T (driver_state) /V () >>
<< /T (driver_postalCode) /V (123456) >>
<< /T (driver_country) /V (RU) >>

]
endobj
trailer
<<
/Root 1 0 R
>>
%%EOF
```

Figure 4. The FDF file produced from the XML file via XSLT

Having determined the names of the FDF fields, we are able to write the XSLT stylesheet that transforms the XML document into the FDF document. Because our XML and FDF formats were completely defined, we only need to create a single XLST stylesheet that can be used at all sites – a significant advantage.

The XSLT stylesheet needs to perform some work other than just wrapping the data from the XML file in angle-brackets and parentheses. The most important two considerations are that strings that contain characters outside the range of U+0000 through U+00FF must be encoded in UTF-16, and that any occurrences of a few special characters (such as parentheses and backslash) must be quoted according to the PDF string quoting rules. The XSLT language provides some string-manipulation methods, but they are not flexible enough to perform UTF-16 quoting easily. (It could probably be done, but we have an easier solution.)

Fortunately, since we are using a Java implementation of the XSLT processor, we can call back into our own Java code from the stylesheet to perform the proper encoding and quoting. To call Java code, we define, in the stylesheet, an XML namespace that

corresponds to a Java class that we write. We can then call methods in our Java class just as we would call the built-in XSLT string methods. Because the XML file in our example is UTF-8 encoded, the strings in the XML file can contain any Unicode text. The XML parser decodes the UTF-8 encoding and passes the text to our method as Java Strings (which are Unicode). See the Java source code in Appendix A for our implementation of a Java method called UTF16StringQuoter that encodes strings in UTF-16 and performs PDF string quoting. Appendix B lists the XSLT stylesheet that uses this method.

The final step is to produce the PDF forms for each legal document to be printed. Any of a number of methods may be used to produce the PDF document, as mentioned above. After producing the basic PDF document, the form fields must be placed into it. There are several methods available to do this, but the most simple is opening the PDF file with Acrobat and using the Form Tool to add form boxes to the document. Figure 5 shows what the PDF form looks like in Acrobat when using the Form Tool.

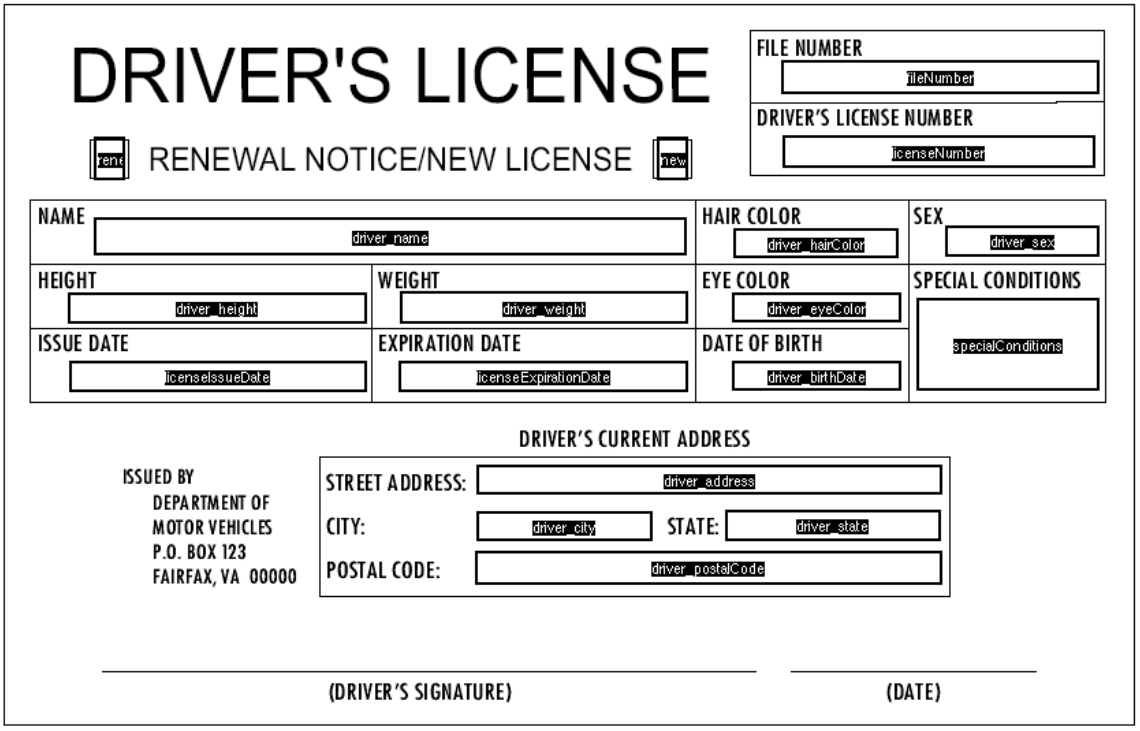

Figure 5. Adding form fields in Acrobat

As the fields are added to the PDF form, they should be given names from the list of field names chosen for the FDF file. Not all the field names that may appear in the FDF file need be used in any form. Instead, the set of field names should be thought of as the collection of available data that can be used to fill in the form.

Effectively, the XSLT stylesheet (the one that transforms the XML file into the FDF file) defines the set of form field names. This set of names can be documented and provided by the software developer to the user who needs to create a PDF form. We provide the list in a set of PDF-form-creating instructions that we include along with our software.

The instructions allow users at the sites where our software is installed to create PDF forms that can produce whatever documents they require.

4.2. Putting it all together

Finally, all these pieces are assembled into the finished software system. Refer to Figure 2, which shows how the documents are transformed through the document generation process. The Java software produces an XML document, and then runs that document through the XSL Transformation process, producing an FDF file. The system then makes the PDF form available in the file system. (We place both the FDF and the PDF file in the operating system's temporary directory, each with temporary names.) Finally, the resulting FDF file is opened with Acrobat Reader, which will locate the associated PDF file and fill in its form fields with the desired data. The result can be seen in Figure 6.

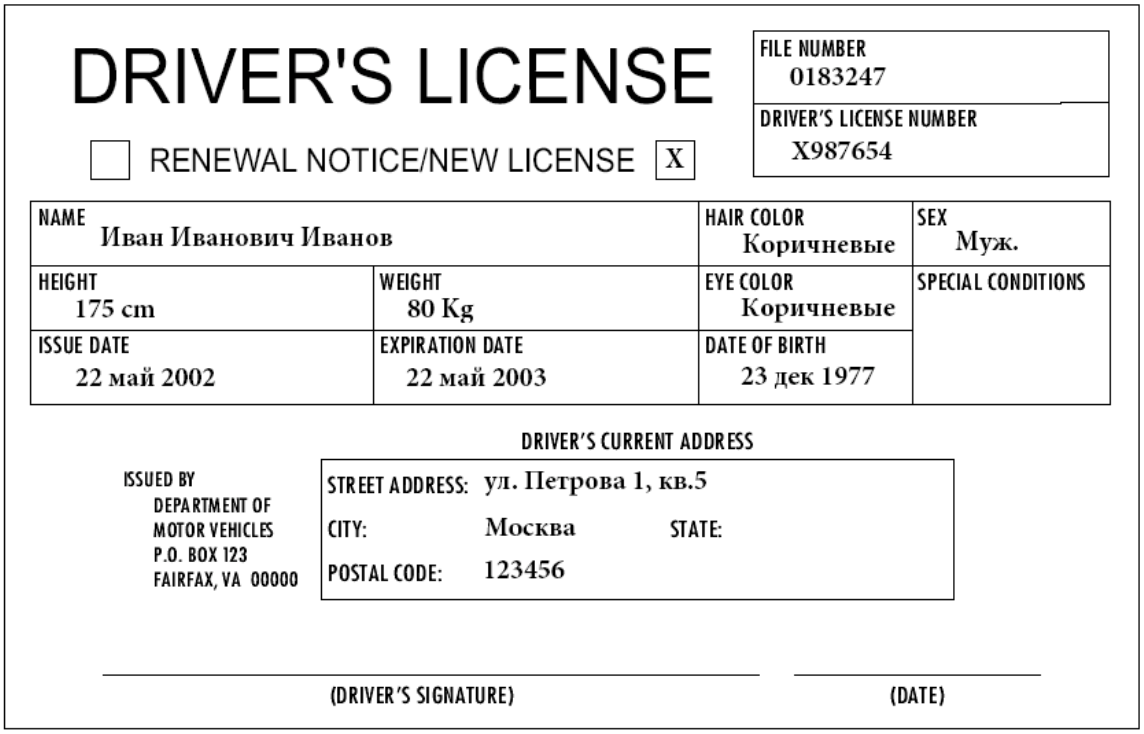

Figure 6. Final filled-in form, obtained by opening the FDF file with Acrobat Reader

4.3. Maintenance issues

As the system has evolved over time, the schema of the XML files produced by the Java code has changed. Sometimes, the hierarchy of data elements is rearranged to allow for more flexibility, but frequently, we just add new information.

This evolution does not cause a problem for the PDF forms, however, because of the insulating qualities of the XSLT transformation step. We have defined a set of form fields that are made available for use in the PDF forms. To maintain that list of fields, all we

have to do is modify the XSLT stylesheet whenever we make a change to the XML file format. Both of these operations are simply part of the development process.

Note that we can add fields to the FDF files in future versions of the software without causing any problems. We simply issue an updated list of the fields available for use in PDF forms, and the sites can either make use of the newly added fields or ignore them.

5. Advanced use: PDF Templates

There are some complex forms that cannot be generated by using a static PDF form document. For instance, a form might have a standard “Part B” page that needs to be attached if some criteria are met on the “Part A” page. Perhaps multiple additional pages are needed.

These more complex forms can be generated by using PDF templates. A PDF template is a special kind of PDF file that contains template pages. Under the instruction of the appropriate FDF file, Acrobat will use these templates to generate pages in a new PDF document in memory. This constructed document is presented to the user as a normal PDF document, and it can be printed the same way. One drawback of this technique is that each user at the site who wants to generate form documents will need to install a full copy of Acrobat; Acrobat Reader cannot generate PDF files from PDF templates.

6. Future

There is an XML-based variant of FDF called XFDF. This file format serves essentially the same purpose as FDF, but it gains the advantages of using XML. In particular, it should be considerably simpler to incorporate Unicode text in the data fields, since XML handles Unicode text natively. This ability should eliminate the need to perform string quoting and UTF-16 encoding.

We have not yet had a chance to experiment with the XFDF format, but the hope is that it will simplify the document generation process.

7. Summary

With the use of Acrobat 5, we have finally implemented a system that meets our goals for producing arbitrary legal documents. Furthermore, we have made the system scalable by allowing each site to create their own forms. Except in the most complex cases (involving techniques such as PDF templates), we can simply provide our software system on a CD, and the users at the site can create the PDF forms to meet their requirements.

APPENDIX A – JAVA UTF-16 STRING QUOTER

```
package com.fgm.util;

import java.text.ParsePosition;
import java.text.SimpleDateFormat;

import java.util.Date;


/**
 * Utility class containing methods useful for dealing with PDF and FDF
 * files
 *
 * @author Geoff Adams
 * @author Xandy Johnson
 * @version $Revision: 1.5 $
 */
public class PDFUtils {

    /**
     * If necessary, encodes characters in a string in UTF-16, with a
     * starting 0xfe 0xff marker.  This is the flag within a PDF string to
     * interpret the byte sequence as a string of UTF-16-encoded
     * characters.  If the String contains no characters outside the
     * Unicode U+0000 – U+00ff range, no UTF-16 transformation is needed,
     * and we simply return the same String, with the exception that some
     * characters are quoted, as required by PDF, and any non-prinable
     * US-ASCII characters are encoded as escaped octals.
     *
     * While this method performs a form of UTF-16 encoding, it is a
     * particular quoted form intended for use in FDF/PDF documents.  The
     * returned String will contain only characters in the range U+0020
     * through U+007D.  It is intended that only the lower octet of each
     * character be used for output into the FDF document.
     *
     * Also note that this method is intended to process an entire PDF
     * string at once.  That is, you should pass it the entire PDF string,
     * from the first character after the opening paren through the last
     * character immediately before the closing paren.  This is because it
     * may UTF-16-encode the string, which PDF requires must be done on a
     * whole string at a time.
     *
     * @param   baseString      The string before any escaping has been
     *                          performed.
     *
     * @return  'baseString' encoded as needed.
     */
    public static String UTF16StringQuoter(String baseString) {

        // First, see whether we need to UTF-16-encode the String,
        // along the way performing any quoting that will be needed if it
        // does not.

        boolean needsEncoding = false;

        // In many cases, the output string will be unchanged from the
        // input string, so preallocate a StringBuffer of the same
        // length.
        StringBuffer buffer = new StringBuffer( baseString.length());

        for ( int i = 0; i < baseString.length(); i++) {
            if (baseString.charAt( i) > 0x00ff) {
                needsEncoding = true;
```

```
                break;
            }
            appendQuotedOctet( buffer, (byte) baseString.charAt( i));
        }
        if (!needsEncoding) {
            return buffer.toString();
        }

        // OK.  We do need to UTF-16 encode it.

        // We'll need at most eight characters to encode each Unicode
        // character, including the beginning Unicode marker.  Preallocate
        // for efficiency.
        buffer = new StringBuffer( (baseString.length() + 1) * 8);

        // Start with the Unicode marker characters
        buffer.append( "\\376\\377");  // 0xfe 0xff

        for ( int i = 0; i < baseString.length(); i++) {

            char character = baseString.charAt( i);

            // This is a hack to work around a bug in Acrobat.  As of
            // Acrobat 5.0.5b CE, Acrobat is still unable to print the S
            // and T characters (either case) with comma below.  Since
            // Romanian text uses these characters, we'll translate them
            // into the similar characters S and T with cedilla.
            // This hack should be removed when this bug is fixed in
            // Acrobat and all relevant sites are updated to the fixed
            // version.
            if (character == 0x0218 || character == 0x219)
                character -= (0x218 - 0x015e);  // Ss comma -> Ss cedilla
            if (character == 0x021a || character == 0x21b)
                character -= (0x21a - 0x0162);  // Tt comma -> Tt cedilla
            // End of Romanian Acrobat hack.

            if (character > 0x0000ffff) {
                // What can we do?!  I haven't implemented handling for
                // Unicode characters greater than U+FFFF
                System.err.println("Ack!  The input string had a 4-byte "
                                   + "Unicode character in it!");
                buffer.append("\\000 ");  // Represent the char with a space
            } else {

                appendQuotedOctet(buffer, (byte) ((character >> 8 ) & 0x00ff));
                appendQuotedOctet(buffer, (byte) (character & 0x00ff));
            }
       }

        return buffer.toString();
    }

    /**
     * Append to the specified StringBuffer the octet value, quoted
     * appropriately for inclusion in an FDF (PDF) document.  The
     * algorithm used here is intended to match the observed behavior in
     * Adobe Acrobat itself.  That is, the octet is represented as a
     * backslash-escaped octal value if it is a non-printable ASCII
     * character, otherwise it is represented as a literal character,
     * unless it is one of \, (, or ), in which case it is
     * backslash-escaped.
     *
     * We take an octet, here, rather than a character, since escaping
     * occurs a byte at a time.  If you need to encode a character whose
     * Unicode value is greater than U+00ff, then you'll need to break the
```

```
     * character into byte-sized chunks, first (and you'll presumably need
     * to encode the entire String as UTF-16).
     *
     * @param   buffer      The StringBuffer to which the
     *                      appropriately-escaped character will be
     *                      appended.
     * @param   octet       The octet to be quoted and appended.
     */
    private static void appendQuotedOctet(StringBuffer buffer, byte octet) {

        if (octet == '\\' || octet == '(' || octet == ')') {

            // PDF requires that we quote \, (, and ) characters with
            // a backslash.
            buffer.append( '\\').append( (char) octet);

        } else if (octet < ' ' || octet > '~' ) {

            // Escape any other non-printable ASCII characters using
            // octal notation.
            buffer.append( escapeOctetOctal( octet));

        } else {

            // Printable ASCII characters are inserted as literals.
            buffer.append( (char) octet);
        }
    }

    /**
     * Return a string which contains the backslash-escaped octal
     * representation of the provided octet.  For example, the octet 0x43,
     * which might represent the ASCII character 'C', would be returned as
     * "\103".
     *
     * @param   octet      The octet to be quoted.
     *
     * @return  The backslash-escaped octal representation of 'octet'.
     */
    public static String escapeOctetOctal(byte octet) {
        return "\\"
            + (char) ( 0x30 + (( octet >> 6 ) & 0x0003) )
            + (char) ( 0x30 + (( octet >> 3 ) & 0x0007) )
            + (char) ( 0x30 + ( octet & 0x0007) );
    }

    /**
     * Formats a date string that is passed in "yyyy-MM-dd" format into the
     * format requested.  Also passes the result through the UTF16StringQuoter
     * because this is intended for use from the stylesheet during document
     * generation, where dates in our XML are in "yyyy-MM-dd" format, but
     * may be desired in other formats for printing.
     *
     * @param   dateStr The string representation of the date in
     *                  "yyyy-MM-dd" format.
     *
     * @param   dateFormat      The format in which to return the date.
     *                          This parameter will be passed to
     *                          java.text.SimpleDateFormat, so please
     *                          refer to that for more details on what
     *                          format strings may be used.
     *
     * @return  'dateStr' in the requested dateFormat
     *
     * @see java.text.SimpleDateFormat
```

```
     * @see #UTF16StringQuoter(String)
     */
    public static String dateFormatter(String dateStr, String dateFormat) {
        // get a date formatter that can parse dates from the format that TXML
        // uses and use it to parse dateStr
        SimpleDateFormat inSdf = new SimpleDateFormat("yyyy-MM-dd");
        Date date = inSdf.parse(dateStr, new ParsePosition(0));

        // get a date formatter for the requested format and use it to format
        // the date
        SimpleDateFormat outSdf = new SimpleDateFormat(dateFormat);
        String formattedDate = outSdf.format(date);

        // pass the results through the UTF16StringQuoter so that the output
        // can be used in an FDF file
        return UTF16StringQuoter(formattedDate);
    }
}
```

APPENDIX B – XSLT STYLESHEET

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:pdfutils="http://www.jclark.com/xt/java/com.fgm.util.PDFUtils"
>

<xsl:output method="text"/>

<!-- We will populate the following FDF fields from data in the XML file:

        (fileNumber)                            (driver_address)
        (licenseNumber)                         (driver_city)
        (licenseIssueDate)                      (driver_state)
        (licenseExpirationDate)                 (driver_postalCode)
        (specialConditions)                     (driver_country)
        (newLicense)  or  (renewal)
                                                (driver_sex)
        (driver_id)                             (driver_height)
        (driver_name)                           (driver_weight)
        (driver_birthDate)                      (driver_hairColor)
                                                (driver_eyeColor)
-->

<!-- Default to referring to a file named license-template.pdf, but if the
invoker passes us a parameter named "template-pdf-file", use its value as
the file name. -->
<xsl:param name="pdf-form-file">driver-license.pdf</xsl:param>

<!-- Transformation from a DriverLicense element in an XML document to an
FDF file follows: -->

  <xsl:template match="DriverLicense">

<!-- the Root-Element pseudo-element is used to anchor the
     beginning of the output, so the "%FDF" starts on line 1,
     character 1. -->

<Root-Element>%FDF-1.2
% Hand-generated via XSLT
1 0 obj
&lt;&lt;
/FDF &lt;&lt; /F (<xsl:value-of select="$pdf-form-file"/>) /Fields 2 0 R >>
>>
endobj
2 0 obj
[
&lt;&lt; /T (fileNumber) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(FileNumber))"/>) >>
&lt;&lt; /T (licenseNumber) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(LicenseNumber))"/>) >>
&lt;&lt; /T (licenseIssueDate) /V (<xsl:value-of
       select="pdfutils:dateFormatter(string(IssueDate),'dd MMM yyyy')"/>) >>
&lt;&lt; /T (licenseExpirationDate) /V (<xsl:value-of
       select="pdfutils:dateFormatter(string(ExpirationDate),'dd MMM yyyy')"/>)
>>
&lt;&lt; /T (specialConditions) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(SpecialConditions))"/>) >>
    <!-- We just need to place an "X" in either the "New" or the "Renewal"
         box on the license.  So, we'll generate an "X" as the value for the
         appropriate field, and leave the other field blank. -->
    <xsl:if test="@type = 'NEW'">
```

```
&lt;&lt; /T (newLicense) /V (X) >>
    </xsl:if>
    <xsl:if test="@type = 'RENEWAL'">
&lt;&lt; /T (renewal) /V (X) >>
    </xsl:if>
<xsl:apply-templates select="Driver"/>
]
endobj
trailer
&lt;&lt;
/Root 1 0 R
>>
%%EOF</Root-Element>

  </xsl:template>

<!--  driver fields  -->

  <xsl:template match="Driver">
&lt;&lt; /T (driver_id) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(ID))"/>) >>
&lt;&lt; /T (driver_name) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(concat(FirstName, ' ', MiddleName, '
', LastName))"/>) >>
&lt;&lt; /T (driver_birthDate) /V (<xsl:value-of
       select="pdfutils:dateFormatter(string(BirthDate),'dd MMM yyyy')"/>) >>
&lt;&lt; /T (driver_height) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(Height))"/>) >>
&lt;&lt; /T (driver_weight) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(Weight))"/>) >>
&lt;&lt; /T (driver_eyeColor) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(EyeColor))"/>) >>
&lt;&lt; /T (driver_hairColor) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(HairColor))"/>) >>
&lt;&lt; /T (driver_sex) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(Sex))"/>) >>

    <xsl:apply-templates select="Address"/>

  </xsl:template>

  <xsl:template match="Address">
&lt;&lt; /T (driver_address) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(StreetAddress))"/>) >>
&lt;&lt; /T (driver_city) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(City))"/>) >>
&lt;&lt; /T (driver_state) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(State))"/>) >>
&lt;&lt; /T (driver_postalCode) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(PostalCode))"/>) >>
&lt;&lt; /T (driver_country) /V (<xsl:value-of
       select="pdfutils:UTF16StringQuoter(string(CountryCode))"/>) >>
  </xsl:template>

</xsl:stylesheet>
```

REFERENCES / BIBLIOGRAPHY

XML

The XML specification is succinct, clear, and well-written, and it includes a number of good examples. It is perhaps the best file format specification I have read.

http://www.w3.org/TR/2000/REC-xml-20001006

XSL (XSLT and FO)

The availability of the Extensible Stylesheet Language as a generic tool is one of the things that make XML so useful as a format. Using XSLT, it is possible to transform data represented in an XML format into almost any other format imaginable.

http://www.w3.org/Style/XSL/

FOP

The project formally known as Formatting Objects to PDF has been renamed Formatting Objects Processor. This is because it is now capable of rendering Formatting Objects into other document types, such as PostScript, PCL, and SVG. Merriam-Webster defines *fop* as "a man who is devoted to or vain about his appearance or dress," and the name of this software is intended to evoke that definition, as well.

http://xml.apache.org/fop/

PDF

Adobe's Portable Document Format has become a de facto standard in many disciplines. The flexibility of the format, the quality of its output, and the availability of readers on a variety of computing platforms have all contributed to its success. Equally important is the fact that the format specification is publicly available.

http://partners.adobe.com/asn/developer/acrosdk/docs.html#filefmtspecs

Acrobat 5 CE and ME

WinSoft produces enhanced versions of several Acrobat products that are more capable of handling the languages of Central Europe and the Middle East than the stock Adobe versions.

http://www.winsoft.fr/

ACKNOWLEDGMENTS

The author would like to acknowledge Keith Bennett, Xandy Johnson, Pete Jones, and Victor Zabolotnyi, coworkers at FGM Inc. who have contributed immensely to the implementation of the software system described in this paper or to the paper itself, as well as the Nonproliferation and Disarmament Fund, U.S. Department of State, which funded the work.