



Professional PHP4 XML

Luis Argerich, Ken Egervari, Matt Anton, Chris Lea, Charlie Killian,
Chris Hubbard, James Fuller



Wrox technical support at: support@wrox.com

Updates and source code at: www.wrox.com

Peer discussion at: p2p.wrox.com

Summary of Contents

Chapter 1:	Introduction	1
Chapter 2:	PHP Fundamentals	11
Chapter 3:	XML Fundamentals	47
Chapter 4:	XML Derivatives	89
Chapter 5:	SAX	129
Chapter 6:	DOM	183
Chapter 7:	XPath	247
Chapter 8:	XSL	283
Chapter 9:	XML Classes	363
Chapter 10:	Putting It Together	391
Chapter 11:	Syndicated Content	439
Chapter 12:	XML to DB, DB to XML	463
Chapter 13:	XML Storage	489
Chapter 14:	PHP As a Client	537
Chapter 15:	SVG	589
Chapter 16:	XML-RPC	631
Chapter 17:	Case Study: A Calendar Server Using XML-RPC	667
Appendix A:	PHP4 XML Language Reference	731
Appendix B:	Installing PHP4 and Apache	747
Appendix C:	SAX 2.0: The Simple API for XML	773
Appendix D:	The XML DOM (Document Object Model)	827
Appendix E:	XSLT Reference	841
Appendix F:	XPath Reference	879
Appendix G:	Object-Oriented Programming	899
Index		919

10

Putting It Together

In the preceding chapters we have taken a detailed look at XML, SAX, DOM, XPath, and XSL. We have also looked at some XML technologies – XHTML, WML, SVG, VXML, RSS, RDF, XML-RPC, and SOAP. Finally, we looked at four XML classes that share one major common feature – they allow us to interact with an XML document.

XML is used for:

- ❑ Storing data – XML documents
- ❑ Describing data – RDF, RSS
- ❑ Querying and Parsing – DOM, SAX, XSLT
- ❑ Presentation – XHTML, WML, SVG, VXML
- ❑ Services – XML-RPC, SOAP

Finding the right mix of XML to use in a project is sometimes a challenging task. This chapter does just that, it describes how to put these different technologies to work.

Here's is a brief summary of what we have learned so far:

- ❑ The basics of XML
- ❑ What is SAX and when and how to use it
- ❑ What is DOM and when and how to use it
- ❑ What is XPath and when and how to use it
- ❑ What XSLT is and when and how to use it

In this chapter we'll be putting what we've learned in the previous chapters into practice to see different solutions to the same problem using different technologies. By the end of this chapter we'll have learned:

- ❑ How to abstract different XML processing problems
- ❑ How to transform XML documents using XSLT or SAX
- ❑ A way to implement SAX filters, and examples of SAX filter chains
- ❑ What can be done to speed up XSLT transformations
- ❑ How to modify XML documents using DOM
- ❑ How to modify XML documents using SAX filters
- ❑ How to modify XML documents using XSLT
- ❑ How to transform XML data into PHP objects
- ❑ How to write XML data from non-XML data using DOM
- ❑ How to write a SAX parser for non-XML data

Programming XML Applications

With a very broad vision we may classify programming tasks using XML into two major categories:

- ❑ Storing and retrieving XML
- ❑ Processing XML

Storing and retrieving XML is detailed in Chapter 13.

Up till now in this book we've covered the core standards for XML processing – XPath, SAX, DOM, and XSLT. In this chapter we are going to study different XML processing problems and study different solutions to these problems using the tools we've mastered in the previous chapters.

We need to process XML data from some source since it is useful to our application. We are going to study some very common problems that arise when processing an XML document. For each problem we'll discuss different options to solve the problem and the scenarios where each solution is better than the others.

The assumption here is that we are in the middle of a large application, and maintaining it is an important issue, hence, we'll be focusing on OOP solutions most of the time. We'll also observe some patterns of OOP that can be applied to programs that process XML.

Transforming XML

When it comes to transforming XML, we have an XML file or XML data source that we need to transform. This can be achieved in many different ways, for example:

- ❑ Transform XML into HTML to be displayed in a browser
- ❑ Transform XML into a PDF file and store it on the disk
- ❑ Transform XML into objects that are going to be stored in an OO database
- ❑ Transform XML into SQL scripts to insert data in a relational database
- ❑ Transform an XML file from a given vocabulary to a different vocabulary
- ❑ Transform XML into proprietary message syntax to be sent over some communication network
- ❑ Transform XML into a text report that will be sent by e-mail
- ❑ Transform XML into an SVG chart to be used in a presentation
- ❑ Transform XML into an XSLT stylesheet that will be used to perform a transformation or validation of another XML document

Transforming implies reading XML, peeking at the content, and writing something that may or may not be XML.

Example

To illustrate different solutions, we'll work with a very simple example. Let's use the following XML file describing some applications coded by a software company:

```
<?xml version="1.0" ?>
<!-- This XML file is an example -->
<apps>
<application id="1">
  <name>Editor</name>
  <author>John</author>
  <bugs>
    <bug>
      <!-- This is one bug -->
      <desc>Foo</desc>
      <sev>2</sev>
    </bug>
    <bug>
      <desc>Bar</desc>
      <sev>3</sev>
    </bug>
  </bugs>
</application>
<application id="2">
  <name>Compiler</name>
```

```
<author>Peter</author>
<bugs>
  <bug>
    <desc>Foo Bar</desc>
    <sev>5</sev>
  </bug>
</bugs>
</application>
</apps>
```

Our transformation task will be to transform all the `<name>` elements to a `<name><b>text</b></name>` format where all the text is uppercase. After transformation, the sample file would look like this:

```
<?xml version="1.0" ?>
<!-- This XML file is an example -->
<apps>
  <application id="1">
    <name><b>EDITOR</b></name>
    <author>John</author>
    <bugs>
      <bug>
        <!-- This is one bug -->
        <desc>Foo</desc>
        <sev>2</sev>
      </bug>
      <bug>
        <desc>Bar</desc>
        <sev>3</sev>
      </bug>
    </bugs>
  </application>
  <application id="2">
    <name><b>COMPILER</b></name>
    <author>Peter</author>
    <bugs>
      <bug>
        <desc>Foo Bar</desc>
        <sev>5</sev>
      </bug>
    </bugs>
  </application>
</apps>
```

We can transform XML documents using:

- ☐ DOM
- ☐ XSLT
- ☐ SAX

Using DOM for XML Transforming

It is possible to use the DOM standard to transform an XML file. We can parse the XML file into the DOM tree and then perform different operations on the tree to transform the source XML file into a different XML file. This is useful only in limited situations, where the result is also XML and the file is small and the transformation simple. Since using the DOM standard for transformation is restricted to very specific situations we shall not look into a detailed example in this chapter.

Here are some of the advantages and disadvantages of using DOM for XML transformations:

Advantages	Disadvantages
Based on a solid standard by the W3C.	Uses a lot of resources. Not good for transforming medium- to large-sized documents.
Easy for removing/adding elements from a document.	Makes sense if the transformation implies modifying a document, very hard if the transformation result is not XML.
	Scales badly, not suitable for online processing unless the document size is known and small.

Using XSLT for XML Transforming

XSLT is the de facto standard for transforming XML content. This language was created to define XML transformations. We write an XSLT stylesheet and then use an XSLT processor to transform the XML file using the XSLT stylesheet, the result can be XML, HTML, a binary format, or some text.

Chapter 8 covers XSLT support in PHP.

We can easily write an XSLT stylesheet to perform our sample transformation. Here is the `tr1.xsl` XSLT stylesheet:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" encoding="utf-8" />
  <xsl:template match="name/text()">
    <b>
      <xsl:value-of select='translate(string(.),
        "abcdefghijklmnopqrstuvwxyz",
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ")' />
    <xsl:apply-templates />
    </b>
  </xsl:template>

  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

The source XML file is transformed using the above stylesheet and the output redirected to a new XML file. Once it is done, we can see that just the text contents of the <name> element are matched, and the output displays the text in uppercase because of the `translate()` function enclosed in a tag in the above stylesheet. We also used a 'match anything else' template to copy the rest of the XML document without modifications to the result.

The presented stylesheet is useful as a base template for changing XML documents since it copies everything that is not matched as-is to the output document.

This is the PHP code used to perform the transformation:

```
<?php
include_once("class_xslt.php");
$xmlslt = new Xslt();
$xmlslt->SetXml("applications.xml");
$xmlslt->SetXsl("tr1.xsl");
if ($xmlslt->Transform()) {
    $ret = $xmlslt->GetOutput();
    print ($ret);
} else {
    print ("Error:". $xmlslt->GetError());
}
?>
```

And this is the `class_xslt.php` abstraction that we used:

```
<?php
class Xslt
{
    var $xsl, $xml, $output, $error ;

    /* Constructor */
    function Xslt()
    {
        $this->processor = xslt_create();
    }

    /* Destructor */
    function Destroy()
    {
        xslt_free($this->processor);
    }

    /* output methods */
    function SetOutput($string)
    {
        $this->output = $string;
    }
    function GetOutput()
    {
        return $this->output;
    }
}
```

```

    }

    /* set methods */
    function SetXmlString($xml)
    {
        $this->xml = $xml;
        return true;
    }

    function SetXslString($xsl)
    {
        $this->xsl = $xsl;
        return true;
    }

    function SetXml($uri)
    {
        if ($doc = new DocReader($uri)) {
            $this->xml = $doc->GetString();
            return true;
        } else {
            $this->SetError("Could not open $xml");
            return false;
        }
    }

    function SetXsl($uri)
    {
        if ($doc = new DocReader($uri)) {
            $this->xsl = $doc->GetString();
            return true;
        } else {
            $this->SetError("Could not open $uri");
            return false;
        }
    }

    /* transform method */
    function Transform()
    {
        $arguments = array('/_xml' => $this->xml,
                           '/_xsl' => $this->xsl
                           );
        $ret = xslt_process($this->processor, 'arg:/_xml', 'arg:/_xsl',
                           NULL, $arguments);
        if (!$ret) {
            $this->SetError(xslt_error($this->processor));
            return false;
        } else {
            $this->SetOutput($ret);
            return true;
        }
    }
}

```

```
/* Error Handling */
function SetError($string)
{
    $this->error = $string;
}
function GetError()
{
    return $this->error;
}
}

/* DocReader -- read a file or URL as a string */
/* test */
/*
    $docUri = new DocReader('http://www.someurl.com/doc.html');
    print ($docUri->GetString());
*/
class DocReader
{
    var $string; // public string representation of file
    var $type;   // private URI type: 'file', 'url'
    var $bignum = 1000000;
    var $uri;

    /* public constructor */
    function DocReader($uri)
    {
                                                // returns integer
        $this->SetUri($uri);
        $this->uri = $uri;
        $this->SetType();

        $fp = fopen($this->GetUri(), "r");
        if ($fp) {
                                                // get length
            if ($this->GetType() == 'file') {
                $length = filesize($this->GetUri());
            } else {
                $length = $this->bignum;
            }
            $this->SetString(fread($fp, $length));
            return 1;
        } else {
            return 0;
        }
    }

    /* determine if a URI is a filename or URL */
    function IsFile($uri)
    {
                                                // returns boolean
        if (strstr($uri, 'http://') == $uri) {
            return false;
        } else {
            return true;
        }
    }
}
```

```

/* set and get methods */
function SetUri($string)
{
    $this->uri = $string;
}
function GetUri()
{
    return $this->uri;
}
function SetString($string)
{
    $this->string = $string;
}
function GetString()
{
    return $this->string;
}
function SetType()
{
    if ($this->IsFile($this->uri)) {
        $this->type = 'file';
    } else {
        $this->type = 'url';
    }
}
function GetType()
{
    return $this->type;
}
}
?>

```

Here we used a PHP class to hide the XSLT processor from the application. This is a key recommendation if we use XSLT. Separating the XSLT implementation from its use has the following advantages:

- ❑ We can change the XSLT processor without changing the code that uses it. For example, we can write a wrapper class for Xalan C and test if Xalan performs worst, better, or equal as compared to Sablotron.
- ❑ If the PHP XSLT extension API changes we won't need to modify all the sources.
- ❑ We can use the class to abstract different sources of XML data for the XSLT transformation. In our class we can pass an XML document as a string or as a file, and by adding some more code, we could have passed a SAX parser, a DOM tree, or a handler to an XML database and performed the transformation there.
- ❑ We can add a caching mechanism to the XSLT class making it transparent to applications. We can write the class to use/not use the caching mechanism according to the user preferences. This allows us to use caching at the level we want if some other class also caches.

It may sound trivial, but abstracting the XML processing tools from the application is a key design decision in a large project. The SAX parser, DOM extension, or XSLT processor should be encapsulated in wrapper classes to ease software maintenance. The PEAR project will surely provide such abstractions in the near future. Stay tuned to <http://pear.php.net/>.

What Happens If the Transformation Is Complex?

If the transformation we need to perform is really complex, then using XSLT is the best option as it can handle complex transformations without any problem. XSLT is rich with features such as the `document()` function, sort capabilities, and features related to writing templates. We can use `<xsl:include>` to modularize our stylesheets, if they get very large. Thus, it is beneficial to use XSLT when we are dealing with large and complex transformations.

What Happens If the XML Document Is Large?

The Sablotron XSLT processor used by PHP parses the whole XML document into memory, as well as the XSLT stylesheet, for processing. This means that the larger the document to transform, the larger will be the memory consumed by the application, and the time that is needed to transform the document.

If the XSLT stylesheet is well-written (XSLT does some things faster than others, for example, expressions tend to be slow) we'll be able to handle transformation of middle- to large-sized documents without much trouble, and we could even do it online.

However, if the documents get really big and the time consumed by the transformation is not acceptable, we can try some of these workarounds:

- ☐ Use a caching mechanism for XSLT transformations
- ☐ Use batch transformations
- ☐ Use SAX

Using a Caching Mechanism for XSLT Transformations

Many times we'll be transforming the same XML document with the same XSLT stylesheet. This is very common, for example, if we use XSLT to create an HTML representation of an XML document for web presentation. It is clear that after the first transformation is done all the others are a waste of resources. Caching implies making the result of XSLT transformations persistent and outputting that result if the same transformation is requested. A caching mechanism can be used in:

- ☐ The XSLT wrapper class used for the XSLT processor
- ☐ A transformation class used to abstract XML transformations
- ☐ The application level

It is advisable that we implement caching at the XSLT processor abstraction level or at the transformation abstraction level, because at the application level our code will be harder to maintain.

A caching mechanism must remember the XML source and the XSLT stylesheet used (storing MD5 checksums is generally a good way to do it), and store the result of the transformation along with the checksums. It also has to decide what policy will be used to prevent the cache from growing ad infinitum – keeping a maximum cache size and cleaning, for example, the 'n' least used transformations, when the cache is full.

With this in mind it will be easy for us to add caching to our transformation classes using, for example, a MySQL database to store the results. Since MySQL is fast, it is a good option for caching. Once caching is enabled we can check if the resources and time used for our transformations are down to acceptable levels. Even if we don't have a performance problem, caching is a good idea as it consumes fewer resources on the server; we don't need them but other processes may.

Using Batch Transformations

If caching is not enough, we must think about not transforming the documents online. We can write a batch transformation system to queue up the transformations and produce the outputs as they are processed.

If we are in an online system this may be difficult, but if we are at the backend of a system we will find that we can transform documents in the background while the application runs in the foreground. This is a very good alternative in the following situations, for example:

- ❑ Transforming XML documents into PDF files that may be downloaded later by the application
- ❑ Transforming XML documents to SQL data that the application may use later
- ❑ Transforming XML documents into reports that will be stored or sent later

However, if we need to use the result of the transformation immediately in the application then this approach is not very useful. Web publishing systems are a clear example of systems that can't benefit from batch transformations.

Batch transformations can be done in many different ways. What we need is a client-server model with a **transformation server**. We store the document and the XSLT stylesheet at some other location and the server does the transformation when available (when not doing other transformations), storing the result at a desired location. Another batch system can then pick up results and put them in the proper place for the application. A database, files, or TCP/IP can be used for communications between the transformation server, and the client application.

The following table shows some of the advantages and disadvantages of using XSLT:

Advantages	Disadvantages
Solid standard by the W3C	May scale badly if the XML documents are really huge
Eases maintainability and reuse of transformations	
Fast enough for online processing of small- to mid-sized documents	
Can be used to perform complex transformations	

Using SAX

If both the above options are not enough, then we can draw the conclusion that an XSLT transformation is not feasible in our application. Though this may happen, it is not a very common scenario. We must be making performing transformations of huge and mutable (non-cacheable) documents for this to happen. If it does happen it is best to use SAX, instead of XSLT, for our transformations. In the next section we will see how to use SAX.

Using SAX for XML Transforming

SAX is an event-driven parser for XML. PHP uses Expat which is a SAX parser written by Jim Clark. In this section we will see how to use SAX for XML transformations.

SAX is described in Chapter 5 of this book.

Why SAX?

SAX breaks the XML file into smaller chunks of data (buffers of raw data) and then parses them, so the amount of memory consumed by SAX is constant, no matter how big the XML file is. This also makes SAX faster than DOM or XSLT, and it scales better than these standards for large files, no matter what we are doing with the document.

How To Use SAX for Transformations

The first intuitive approach for using SAX to transform XML documents is to write handlers for the SAX events, and, keeping context between the handlers, perform the transformation inside them. In our example we can think about the following solution:

- ❑ Step 1 – Initialize a flag to 0
- ❑ Step 2 – Define the handlers for SAX events:
 - ❑ `StartElementHandler`
If the element is `<name>` turn a flag on (1)
 - ❑ `EndElementHandler`
If the element is `<name>` turn the flag off (0).
 - ❑ `CharacterDataHandler`
If the flag is on (1) then output uppercased data, else output data
- ❑ Step 3 – Parse the document

We are assuming that no subelements may appear inside a `<name>` tag (no mixed content). If subelements do appear then we can use an integer `flag` and increment or decrement it. This way we can keep track of the text of a `<name>` tag and not its subelement `<name>`.

We just wrote the sketch of a SAX parser that transforms our XML file. We can easily write this using the PHP XML parser functions, as we learned in Chapter 5, and the result will be the desired transformation. While this works, it is not a very good long-term solution.

After writing applications that use SAX we'll learn that complex processing becomes a nightmare due to the limitations of the SAX parser. We will not feel very comfortable if the parser only tells us when an element starts, or when an element ends. As processing becomes more complex we will have to write a lot of code to keep context using stacks, push and pull flags, use integer semaphores, and many other obscure programming techniques.

Imagine a situation where we have to perform a complex transformation. It may be the worst coding nightmare ever, and even if we succeed, it will still be difficult for us to understand the code after some weeks. Every programmer has to be extra careful when faced with such situations. As programmers we know that the best way to address a complex problem is to break it into pieces, solve each piece, and then unite all of them in the end. This is called **modularization** or factoring. Modularization of SAX driven applications is crucial whenever the task is complex. We can do this using **SAX filters**.

SAX Filters for XML Transformations

We have learnt a lot about SAX and filters in Chapter 5. In this chapter we touch on this concept of filters, and some techniques to use them. It is expected that PEAR will define a standard for SAX filters in PHP in the near future. In the meantime we can use the approach discussed in Chapter 5, or our own approach, provided we understand the concept well.

A SAX filter is a class that performs transforming from the output of a SAX parser or another SAX filter. A filter can encapsulate a simple transformation, so we can write many filters and **chain** them later. Passing the output of one filter as an input to another creates a complex transformation from simple ones.

We'll be building a very simple example to illustrate these concepts. Let's now write the class.

The `AbstractSAXParser` receives an XML file and produces SAX events that must be passed to a filter. The SAX parser alone is useless: it is used as a generator of SAX events for another class.

The abstract class is important as it helps us to understand the methods that implementations must have.

Here are some observations:

- ❑ **The class hides the SAX parser from the application**
If we change the parser from Expat to another one, nothing will have to be changed except this class.
- ❑ **The class allows parsing non-XML data**
We can write an `AbstractSAXParser` for non-XML data, parse the data with it, and produce SAX events. This way we can transform non-XML to XML.
- ❑ **The class can hide the XML source from its users**
We can write a SAX parser for XML files, XML strings, and for XML stored in a database.

The abstract class is as follows:

```
class AbstractSAXParser
{
    var $listener;

    function AbstractSAXParser() {}

    function ParserSetOption($opt, $val) {}

    function SetListener($obj)
    {
        $this->listener = $obj;
    }
}
```

```

    }

    function StartElementHandler($parser, $name, $attrs)
    {
        $this->listener->StartElementHandler($name, $attrs);
    }

    function EndElementHandler($parser, $name)
    {
        $this->listener->EndElementHandler($name);
    }

    function CharacterDataHandler($parser, $data)
    {
        $this->listener->CharacterDataHandler($data);
    }

    function Parse() {}
}

```

The **listener** will be the object that will receive the SAX events produced by this class. As we can see, the handlers in this class are fixed, and they only call their brothers in the listener object. The **abstract parse method** will depend on the implementation, and is supposed to parse the XML data. It generates the events that are intercepted by the methods in this class and calls the same methods in the listener. Note that the listener methods don't receive a parser since they don't need it. The `SetListener()` method allows us to set the object that will be used as a listener for the events produced in this class. Finally the `ParserSetOption()` method is prepared to set options specific to the SAX parser, such as case folding.

Note that for space-saving reasons only `startElement`, `endElement`, and `characterData` events are considered. It is easy to extend the example to the full range of SAX events with very minor changes.

We can write a class for the Expat parser, as follows:

```

class ExpatParser extends AbstractSAXParser
{
    var $parser;
    var $filename;
    var $buffer;
    var $error_string;
    var $line;

    function ExpatParser($xmlfile)
    {
        $this->filename = $xmlfile;
        $this->parser = xml_parser_create();
        $this->buffer = 4096;
        xml_set_object($this->parser, &$this);
        xml_set_element_handler($this->parser, "StartElementHandler",

```

```

        "EndElementHandler");
        xml_set_character_data_handler($this->parser,
                                      "CharacterDataHandler");
    }

    function ParserSetOption($opt, $val)
    {
        return xml_parser_set_option($this->parser, $opt, $val);
    }

    function Parse()
    {
        if (!$fp = fopen($this->filename, "r")) {
            return 0;
        }

        while ($data = fread($fp, $this->buffer)) {
            if (!xml_parse($this->parser, $data, feof($fp))) {
                $this->error_string =
                    xml_error_string(xml_get_error_code($xml_parser));
                $this->line = xml_get_current_line_number($xml_parser);
                die("Error: ".$this->error_string." on ".$this->line);
            }
        }
        xml_parser_free($this->parser);
        return 1;
    }
}

```

This class is just an implementation of the abstract class using the Expat parser. Some methods such as `getErrorString()` and `getErrorLineNumber()` are missing, but we won't need them for our examples.

Now we define an `AbstractFilter` class. The abstract filter receives SAX events, manipulates them, and re-transmits the events to a listener. Also, the filter can change tag names, modify the content of elements, add elements by generating events, and so on.

Here is the abstract class:

```

class AbstractFilter
{
    var $listener;

    //to set the listener of the filter and handlers for the events
    function SetListener($obj)
    {
        $this->listener = $obj;
    }

    function StartElementHandler($name, $attrs) {}

    function EndElementHandler($name) {}

    function CharacterDataHandler($data) {}
}

```

The `SetListener()` method sets the listener of the filter and the handlers for the events. In the handlers we have to perform a task and then pass the event to the next listener.

We will use the same example that we used earlier in this chapter, and write a filter to convert the text inside `<name>` tags to uppercase:

```
class FilterName extends AbstractFilter
{
    var $flag = 0;

    function StartElementHandler($name, $attribs)
    {
        if (strtolower($name) == "name") {
            $this->flag = 1;
        } else {
            $this->flag = 0;
        }
        $this->listener->StartElementHandler($name, $attribs);
    }

    function EndElementHandler($name)
    {
        if (strtolower($name) == "name") {
            $this->flag = 0;
        }
        $this->listener->EndElementHandler($name);
    }

    function CharacterDataHandler($data)
    {
        if ($this->flag) {
            $data = strtoupper($data);
        }
        $this->listener->CharacterDataHandler($data);
    }
}
```

Note that we used the simple approach we described before, that is, using a `flag` to track when we are inside a `<name>` element and when we are not. Also notice how the listener is passed the events immediately after the modification is done. The listener will receive the same events as this filter but it will receive upcased text for `<name>` elements.

Finally we can write a filter that doesn't produce events but outputs the XML for testing:

```
class FilterOutput extends AbstractFilter
{
    function StartElementHandler($name, $attribs)
    {
        print("<".$name);
        foreach ($attribs as $key => $val) {
            print(" $key = '$val'");
        }
        print("><br />");
    }

    function EndElementHandler($name)
```

```

    {
        print ("&lt;/$name&gt;<br />");
    }

    function CharacterDataHandler($data)
    {
        print ("$data");
    }
}

```

As we can see this filter can't be used in the middle of a chain because it consumes all the SAX events. It just outputs the XML data (the `print()` statements where it used to just write a short example).

This is the way we can use them:

```

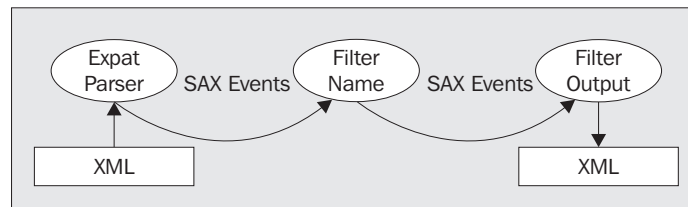
$f1 = new ExpatParser("applications.xml");
$f1->ParserSetOption(XML_OPTION_CASE_FOLDING, 0);
$f2 = new FilterName();
$f3 = new FilterOutput();

$f2->SetListener($f3);
$f1->SetListener($f2);

$f1->Parse();

```

It is very important to build the chain in the proper order. We first have to set the listener of the `$f2` filter as `$f3` and then the listener of `$f1` as `$f2`. The following figure shows what will happen:



If we remember correctly, our transformation not only asked us to uppercase the `<name>` tags but also to place them in a `<b>` tag. We can write a filter that converts `<name>some</name>` elements into `<name><b>some</b></name>` elements as follows:

```

class FilterNameBold extends AbstractFilter
{
    var $flag = 0;

    function StartElementHandler($name, $attribs)
    {
        if (strtolower($name) == "name") {
            $this->flag = 1;
        } else {
            $this->flag = 0;
        }
        $this->listener->StartElementHandler($name, $attribs);
    }
}

```

If we are in a <name> element (`flag = 1`) then we generate an artificial `startElement` event with a `b` tagname by calling the `StartElementHandler()` method in the listener:

```
        if ($this->flag) {$this->listener->
            StartElementHandler("b", array());}
    }

    function EndElementHandler($name)
    {
        if (strtolower($name) == "name") {
            $this->flag = 0;
            $this->listener->EndElementHandler("b");
        }
        $this->listener->EndElementHandler($name);
    }

    function CharacterDataHandler($data)
    {
        if ($this->flag) {
            $data = strtoupper($data);
        }
        $this->listener->CharacterDataHandler($data);
    }
}
```

This is a nice example because it uses a common trick in SAX filters. To add an element we just invent an event for that element by passing it to the next filter. We can run the example instantiating the new chain of filters as follows:

```
$f1 = new ExpatParser("applications.xml");
$f1->ParserSetOption(XML_OPTION_CASE_FOLDING, 0);
$f2 = new FilterNameBold();
$f2b = new FilterName();
$f3 = new FilterOutput();

$f2b->SetListener($f3);
$f2->SetListener($f2b);
$f1->SetListener($f2);

$f1->Parse();
```

The result of this will be the XML file with composite transformation applied.

Streaming

One good characteristic of SAX filters is that as the SAX parser parses the XML chunks, SAX events are propagated for the chunk. So the last filter in the chain can stream the result of the composite transformation step-by-step without having to concatenate or wait for the whole result to arrive. This allows streaming the result to the disk, database, or whatever its destination is without delays.

To stream the result we need to have a streaming filter as the last link in the chain that, instead of calling the `print()` statements in our example, calls a `Stream()` method in another object. That object can save the results chunk by chunk to a file, database, or send them to the browser:

```

class FilterStreamer extends AbstractFilter
{
    var $stream;

    function FilterStreamer($obj)
    {
        $this->stream = $obj;
    }

    function StartElementHandler($name, $attrs)
    {
        $this->stream->Stream("<$name");
        foreach ($attrs as $key => $val) {
            $this->stream->Stream(" $key='$val'");
        }
        $this->stream->stream(">");
    }

    function EndElementHandler($name)
    {
        $this->stream->Stream("</$name>");
    }

    function CharacterDataHandler($data)
    {
        $this->stream->Stream("$data");
    }
}

```

This filter expects to receive an object in its constructor, and that object must implement a `Stream()` method capable of receiving the result of the filter chain as chunks. For example:

```

class SimpleStreamer extends AbstractStreamer
{
    function Stream($data)
    {
        print ($data);
    }
}

```

We can build the filter chain as follows:

```

$ss = new SimpleStreamer();

$f1 = new ExpatParser("applications.xml");
$f1->ParserSetOption(XML_OPTION_CASE_FOLDING, 0);
$f2 = new FilterNameBold();
$f3 = new FilterStreamer($ss);
$f2b = new FilterName();

$f2b->SetListener($f3);
$f2->SetListener($f2b);
$f1->SetListener($f2);

$f1->Parse();

```

The following table shows some of the advantages and disadvantages of using SAX:

Advantages	Disadvantages
Scales very well for huge documents	Transformations can be complex
Resource consumption is constant for documents of any size	Harder to use if the result is not XML
It is fast	Requires a lot of coding for even simple transformations
Allows streaming	

Choosing a Transformation Strategy

So far in this chapter we've seen how DOM, XSLT, and SAX can be used to perform transformations. The question is when to use each approach, and as we can imagine there's no definite answer. The best recommendation, however, seems to be to use XSLT whenever we can and SAX only when XSLT can't be used. DOM could be an alternative only for very simple transformations.

This table describes the best possible approach in each situation:

Document Type	Simple Transformations	Medium Complexity Transformations	Complex Transformations
Small	DOM or XSLT	XSLT	XSLT
Mid-sized	XSLT or DOM (if the transformation is very simple)	XSLT	XSLT
Large	XSLT or SAX	XSLT (preferred) or SAX	XSLT
Very large	SAX	SAX or XSLT	Try to use XSLT or SAX (if XSLT is impossible)

If transformations are almost always the same, we may want to use XSLT and caching no matter how big the documents are, since we'll only waste time and resources once in a while.

If the result of the transformation is not XML, XSLT is the best alternative and we should try to evade DOM or SAX unless there's no escape.

Abstracting XML Transformations

As we can see, sometimes the choice between SAX and XSLT can be difficult, and at times we may start using XSLT and then need to prepare a SAX alternative for huge documents. Taking this into consideration we recommend abstract transformations by defining transformation classes that implement a given transformation, for example, RSS to HTML. We then can implement that transformation class using XSLT or SAX without changing the application. The class is also a good place to put a caching mechanism if necessary. Abstraction is always worth the effort.

Modifying XML

Modifying XML is a problem where we have an XML data source and need to modify it. We may want to change some text, add an attribute, add an element, or remove a subtree of the document.

Modifying is a subset of the transforming problem since the output is always XML, and belongs to the same vocabulary as the original file.

Here are some situations where we might find instances of this problem:

- ❑ Using XML data as a catalog and then having to add/modify/remove items from the catalog
- ❑ Using XML to keep statistics that need to be updated
- ❑ Using XML for logging and having to add data

Example

For example, let's imagine we use XML to keep track of a shopping cart in some online store application. We may face a simple XML such as:

```
<cart>
  <user uid="382378">
    <name>John Smith</name>
  </user>
  <products>
    <product pid="12">
      <name>ACMEPencilHB</name>
      <desc>Soft Pencil by ACME</desc>
      <quantity>2</quantity>
      <unit_price>0.50</unit_price>
    </product>

    <product pid="13">
      <name>ACMEPen089</name>
      <desc>Rollerball pen</desc>
      <quantity>1</quantity>
      <unit_price>1</unit_price>
    </product>

  </products>
</cart>
```

Customers will use an interface to add/remove products from the shopping cart, and we have to reflect those changes to the XML file. Therefore, we'll need a way to:

- ❑ **Add a product**
Given the `pid` (product ID), `<name>`, `<desc>`, `<quantity>`, and `<unit_price>`. We'll add the product to the cart; if the product already exists in the cart we don't change it, we add it to the end.
- ❑ **Remove a product**
Given the `pid`, we remove all instances of the product from the cart.

Of course we should need more functions for our shopping cart to be usable, but let's use only these three for our examples.

When modifying XML is our task, the options are:

- ❑ DOM
- ❑ SAX
- ❑ XSLT

Using DOM To Modify XML Files

DOM is a good tool to modify XML files since it represents the XML document as an object and has all the methods we need to modify the object. We have methods to add/remove elements, add/remove attributes, move subtrees, and many other tasks. All this would be hard to code using XSLT or SAX.

While DOM eases the process, we have to remember that it consumes a lot of resources and that sometimes, if the documents are really big, it won't perform well enough. Only when the DOM approach is failing due to the size of the XML file, we'll have to look for an XSLT- or SAX-based solution.

For our example we have to write a function to add a product to the XML shopping cart and a function to remove a product from the cart. This is the `addProduct()` function:

```
function AddProduct($xml, $id, $name, $desc, $quantity, $unit_price)
{
    $doc = xmldoc($xml);
    //$doc = domxml_open_mem($xml);

    $root = $doc->document_element();
    $children = $root->child_nodes();
    foreach ($children as $child) {
        if ($child->node_type() == XML_ELEMENT_NODE){
            if ($child->tagname() == "products") {
                $newpro = $doc->create_element("product");
                $newpro->set_attribute("pid", $id);

                $newname = $doc->create_element("name");
                $newname_text = $doc->create_text_node($name);
                $newname->add_child($newname_text);
```

```

        $newprice = $doc->create_element("unit_price");
        $newprice_text = $doc->create_text_node($unit_price);
        $newprice->add_child($newprice_text);

        $newdesc = $doc->create_element("desc");
        $newdesc_text = $doc->create_text_node($desc);
        $newdesc->add_child($newdesc_text);

        $newquan = $doc->create_element("quantity");
        $newquan_text = $doc->create_text_node($quantity);
        $newquan->add_child($newquan_text);

        $newpro->add_child($newname);
        $newpro->add_child($newprice);
        $newpro->add_child($newdesc);
        $newpro->add_child($newquan);

        $child->add_child($newpro);

    }
}
$xml = $doc->dump_mem();

return $xml;
}

```

Note how the DOM tree is traversed looking for the <products> element, and then the `add_child()`, `create_text_node()`, and `create_element()` methods are used to create a <product> element and its subelements. The `set_attribute()` method is also used to set an attribute for the <product> element. This mechanism of navigation and insertions is very common when using DOM.

`add_child()` is an alias of `append_child()`. `append_child()` is the proper function to use and works in PHP 4.2.1 and above. So use `append_child()` instead of `add_child()` with newer versions of PHP.

To remove all the instances of a product we can use this function:

```

function RemoveProduct($xml, $id)
{
    $doc = xmldoc($xml);
    //$doc = domxml_open_mem($xml);

    $root = $doc->root();
    $children = $root->child_nodes();
    foreach ($children as $child) {
        if ($child->node_type() == XML_ELEMENT_NODE){
            if ($child->tagname() == "products") {
                $products = $child->child_nodes();
            }
        }
    }
}

```

```

        foreach ($products as $product) {
            if ($product->node_type() == XML_ELEMENT_NODE) {
                // $product_children = $product->child_nodes();
                $ats = $product->get_attribute("pid");
                if ($ats == $id) {
                    $product->unlink_node();
                }
            }
        }
    }
}
$xml = $doc->dumpmem();
return $xml;
}

```

The function shows how to iterate the DOM tree looking for <product> tags, and how to check for an attribute of an element to see if we did find a product with the ID that we passed to the function.

To be more efficient we may write functions to add/remove elements from the DOM tree, thus parsing the document only once even if we want to add and remove many products. We didn't use this approach on other examples, but it is usually recommended if possible.

DOM is a solid standard for XML modification with methods to add/remove and change nodes. However, it is still not scalable in PHP, and especially scales badly for large XML documents. It also consumes lots of resources.

If you get an error such as 'undefined function xxx' when trying to test the examples in this chapter, you are probably using an old version of PHP. Examples in this section have been produced from the new DOM extension in PHP 4.2. If you have an earlier version you may want to check the following list of deprecated methods and their replacements in 4.2.

From the PHP documentation:

Deprecated Function	Replacement Function
DomDocument_dtd()	DomDocument_doctype()
DomDocument_root()	DomDocument_document_element()
DomDocument_children()	DomNode_child_nodes()
DomDocument_imported_node()	No replacement
DomNode_add_child()	Create a new node with, for example, DomDocument_create_element() and add it with DomNode_append_child()
DomNode_children()	DomNode_child_nodes()

Deprecated Function	Replacement Function
<code>DomNode_parent()</code>	<code>DomNode_parent_node()</code>
<code>DomNode_new_child()</code>	Create a new node with, for example, <code>DomDocument_create_element()</code> and add it with <code>DomNode_append_child()</code>
<code>DomNode_set_content()</code>	Create a new node with, for example, <code>DomDocument_create_text_node()</code> and add it with <code>DomNode_append_child()</code> .
<code>DomNode_get_content()</code>	Content is just a text node and can be accessed with <code>DomNode_child_nodes()</code>
<code>DomNode_set_content()</code>	Content is just a text node and can be added with <code>DomNode_append_child()</code>

Using SAX To Modify a Document

If we could use SAX to transform a document we can surely use it to modify it. In fact, the approach used to modify a document with SAX should be the same as used when transforming the document – write filters that perform some basic transformations, and then chain the filters if needed, to perform a more complex modification. When modifying a document, chaining will be less frequent but the filter approach is good because each transformation is a separate class that can be maintained and updated easily.

Using SAX filters we can write the following filter to add a product to our XML cart example:

```
class FilterAddProduct extends AbstractFilter
{
    var $products = Array();

    function AddProduct($id, $name, $desc, $quantity, $unit_price)
    {
        $selem = Array();
        $selem["pid"] = $id;
        $selem["name"] = $name;
        $selem["desc"] = $desc;
        $selem["quantity"] = $quantity;
        $selem["unit_price"] = $unit_price;
        $this->products[] = $selem;
    }

    function StartElementHandler($name, $attrs)
    {
        $this->listener->StartElementHandler($name, $attrs);
        if ($name == "products") {
            // Perform insertions
            foreach ($this->products as $product) {
                $this->listener->StartElementHandler("product",
                    Array("pid" => $product["pid"]));
            }
        }
    }
}
```

```

        // Name
        $this->listener->StartElementHandler("name", Array());
        $this->listener->CharacterDataHandler($product["name"]);
        $this->listener->EndElementHandler("name");

        // Desc
        $this->listener->StartElementHandler("desc", Array());
        $this->listener->CharacterDataHandler($product["desc"]);
        $this->listener->EndElementHandler("desc");

        // Quantity
        $this->listener->StartElementHandler("quantity", Array());
        $this->listener->CharacterDataHandler($product["quantity"]);
        $this->listener->EndElementHandler("quantity");

        // Unit_price
        $this->listener->StartElementHandler("unit_price", Array());
        $this->
            listener->characterDataHandler($product["unit_price"]);
        $this->listener->EndElementHandler("unit_price");

        $this->listener->EndElementHandler("product");
    }
}

function EndElementHandler($name)
{
    $this->listener->EndElementHandler($name);
}

function CharacterDataHandler($data)
{
    $this->listener->CharacterDataHandler($data);
}
}

```

This filter lets us prepare the filter to add many products when parsing the document; we store the products to be added in a member array of the filter. When parsing the document, if the filter sees that a `<products>` element has started, it traverses the list of products to be added, generating the events needed to insert the elements on the XML file.

We can use the filter this way:

```

$f1 = new ExpatParser("cart.xml");
$f1->ParserSetOption(XML_OPTION_CASE_FOLDING, 0);

$f2 = new FilterAddProduct();
$f2->AddProduct("15", "foo", "foo bar", "1", "6");
$f2->AddProduct("17", "Goo", "GOO", "2", "16");

$f3 = new FilterOutput();
$f2->SetListener($f3);
$f1->SetListener($f2);

$f1->Parse();

```

To remove elements from an XML document using SAX, we can use the following filter:

```
class FilterRemoveProduct extends AbstractFilter
{
    var $products = Array();
    var $ignore = 0;

    function RemoveProduct($id)
    {
        $this->products[] = $id;
    }

    function StartElementHandler($name, $attrs)
    {
        if ($name == "product") {
            if (in_array($attrs["pid"], $this->products)) {
                $this->ignore = 1;
            } else {
                $this->listener->StartElementHandler($name, $attrs);
            }
        }
        if (!$this->ignore) {
            $this->listener->StartElementHandler($name, $attrs);
        }
    }

    function EndElementHandler($name)
    {
        if (!$this->ignore) {
            $this->listener->EndElementHandler($name);
        } else {
            if ($name == "product") {
                $this->ignore = 0;
            }
        }
    }

    function CharacterDataHandler($data)
    {
        if (!$this->ignore) {
            $this->listener->CharacterDataHandler($data);
        }
    }
}
```

This filter can also store IDs of products to be deleted; we can use the `removeProduct()` method as many times as we want before parsing the document, preparing our chain to remove as many elements as we want.

We use a flag in the filter, and we turn the flag on each time a `<product>` element with an ID to be deleted starts. While the flag is on, we don't propagate events to the following filter, thus eliminating the element's text and subelements. When a `<product>` element ends, we turn the flag off (if it was on), to let further products pass to the filter. Ignoring SAX event propagation is a common way to eliminate elements from an XML document using SAX.

In the following example we add some products to the cart and then remove some others:

```
$f1 = new ExpatParser("cart.xml");
$f1->ParserSetOption(XML_OPTION_CASE_FOLDING, 0);
$f2 = new FilterAddProduct();
$f2->AddProduct("15", "foo", "foo bar", "1", "6");
$f2->AddProduct("17", "Goo", "GOO", "2", "16");

$f3 = new FilterRemoveProduct();
$f3->RemoveProduct("13");
$f3->RemoveProduct("15");

$f4 = new FilterOutput();

$f3->SetListener($f4);
$f2->SetListener($f3);
$f1->SetListener($f2);

$f1->Parse();
```

SAX scales very well for huge documents. Its resource consumption is constant for documents of any size, and it is fast. However, every modification needs coding, and at times it is complex. The same modifications are easy using DOM.

Using XSLT To Modify a Document

XSLT can also be used to modify an XML document; we need to write stylesheets to perform the transformations, receiving parameters about what to do.

For example, we can write the following XSLT file to add a product to a shopping cart:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" encoding="utf-8" />

<xsl:param name="pid" />
<xsl:param name="name" />
<xsl:param name="desc" />
<xsl:param name="quantity" />
<xsl:param name="unit_price" />

<xsl:template match="/cart/products">
  <xsl:element name="product">
    <xsl:attribute name="pid">
      <xsl:value-of select="$pid" />
    </xsl:attribute>
    <xsl:element name="name"><xsl:value-of select="$name" /></xsl:element>
    <xsl:element name="desc"><xsl:value-of select="$desc" /></xsl:element>
    <xsl:element name="quantity">
      <xsl:value-of select="$quantity"/>
    </xsl:element>
    <xsl:element name="unit_price">
      <xsl:value-of select="$unit_price"/>
    </xsl:element>
  </xsl:element>
</xsl:template>
<xsl:apply-templates />
```

```

</xsl:template>

<xsl:template match="@*|node()">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()" />
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>

```

The interesting issue here is that the XSLT stylesheet receives parameters with the information of the product to be added. The parameters declared at the beginning of the stylesheet show the parameters that the stylesheet is expecting to receive.

This PHP file shows how to pass parameters to an XSLT stylesheet using Sablotron:

```

// Allocate a new XSLT processor
$xh = xslt_create();

$new_prod = Array("pid" => "19",
                  "name" => "Newby",
                  "desc" => "A new product",
                  "quantity" => "1",
                  "unit_price" => "19"
                  );

$result = xslt_process($xh, 'cart.xml',
                      'addProduct.xsl', NULL, Array(), $new_prod);

if ($result) {
    print ($result);
} else {
    print ("Sorry, cart.xml could not be transformed by ");
    print ("addProduct.xsl into the \$result variable. ");
    print ("The reason is that " . xslt_error($xh));
    print (" and the error code is " . xslt_errno($xh));
}

xslt_free($xh);

```

We are not using an XSLT abstraction class here to show how to pass parameters to Sablotron.

And the following XSLT stylesheet removes products from the cart:

```

<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml" encoding="utf-8" />
<xsl:param name="pid" />
  <xsl:template match="/cart/products/product[@pid=$pid]">
  </xsl:template>

  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>

```

This is a simple stylesheet where, if the product with the ID passed as a parameter is found, then nothing is sent and everything else is copied.

This PHP code tests the stylesheet:

```
// Allocate a new XSLT processor
$xh = xslt_create();

$new_prod = Array("pid" => "13");

$result = xslt_process($xh, 'cart.xml',
                      'removeProduct.xsl', NULL, Array(), $new_prod);

if ($result) {
    print $result;
} else {
    print ("Sorry, cart.xml could not be transformed by ");
    print ("removeProduct.xsl into the \"$result\" variable. ");
    print ("The reason is that " . xslt_error($xh));
    print (" and the error code is " . xslt_errno($xh));
}

xslt_free($xh);
```

If the DOM approach is too slow or uses a lot of resources, or we find it hard to write SAX filters for some modifications, it is good to use an XSLT stylesheet.

The main advantage of using XSLT is that powerful and complex modifications can be performed. However, it uses more resources than SAX, and passing parameters is not so easy if we have to perform complex transformations. One more problem with XSLT is that it does not have a caching option as yet.

Abstracting the Modifications To an XML Document

As with transformations, we may implement modifications to an XML document in many different ways – using XSLT, SAX filters, or the DOM approach as per the circumstances. Then it's logical to think that abstracting the modifications can be a good idea. We have several ways to abstract transformations to an XML document – the command pattern of OOP can be used to implement a command processor that can perform/undo commands from an XML document.

We can find a good description of the command pattern in this article from <http://www.xml.com/pub/a/2000/01/19/feature/index.html>.

Creating Objects from XML Files

This problem is stated as follows – we have an XML document and we want to create PHP objects from the information in the document. These objects can then be used for different tasks depending on the application that needs them. This is known as the **XMLable** pattern.

Example

We may have an XML file describing books, and we may want to create book objects from our XML document. This is a simplified representation of the XML document:

```
<?xml version="1.0"?>
<books>
  <book>
    <title>The Two Tours</title>
    <author>J.R.R Tolkien</author>
  </book>

  <book>
    <title>Second Foundation</title>
    <author>I.Asimov</author>
  </book>

  <book>
    <title>Foundation and Empire</title>
    <author>I.Asimov</author>
  </book>
</books>
```

And this can be the Book class:

```
class Book
{
    var $title;
    var $author;

    function GetTitle()
    {
        return $this->title;
    }

    function GetAuthor()
    {
        return $this->author;
    }

    function SetTitle($title)
    {
        $this->title = $title;
    }

    function SetAuthor($author)
    {
        $this->author = $author;
    }
}
```

As usual we may think about DOM, XSLT, or SAX, but in this problem we don't have choices, let's see why.

Firstly, DOM is unacceptable in this case. We parse the XML document into the DOM tree, and then we'll be facing the same problem we had before – how to convert the DOM tree to many book objects. XSLT is too exotic for this problem; we may write an XSLT transformation that creates PHP code that creates the objects, and then `eval()` the transformation result.

So the obvious solution is to use SAX, where we parse the XML document and we create the objects as we collect enough information for each one of them.

Using SAX To Create PHP Objects from XML

Here we build a SAX filter that can be used to create the book objects from our document. We store references to the objects in an array to keep track of the objects in memory. This is the strategy used to create objects:

Whenever a `<book>` tag is found, create a new book object and store the reference in the array. Whenever an `<author>` or a `<title>` tag is found, store the content in the object property with the same name.

Here is the code:

```
class FilterBuildBooks extends AbstractFilter
{
    var $oneBook = 0;
    var $book;
    var $property = 0;
    var $books = Array();

    function GetBooks()
    {
        return $this->books;
    }

    function StartElementHandler($name, $attrs)
    {
        if ($name == "book") {
            $this->books[] = new Book();
            $this->oneBook = 1;
        } else {
            if ($this->oneBook) {
                $this->property = $name;
            }
        }
        $this->listener->StartElementHandler($name, $attrs);
    }

    function EndElementHandler($name)
    {
        $this->property = 0;
        if ($name == "book") {
            $this->oneBook = 0;
        }
        $this->listener->EndElementHandler($name);
    }
}
```

```

    }

    function CharacterDataHandler($data)
    {
        if ($this->oneBook && $this->property) {
            // Here we construct the name of the method to be called.
            $method_get = "get".strtoupper(substr($this->property, 0, 1)) .
                substr($this->property, 1);
            $method_set = "set".strtoupper(substr($this->property, 0, 1)) .
                substr($this->property, 1);
            $prop = $this->books[count($this->books)-1]->$method_get();
            $this->books[count($this->books)-1]->$method_set($prop.$data);
        }
        $this->listener->CharacterDataHandler($data);
    }
}

```

This PHP code parses the XML file that creates the book objects:

```

$f1 = new ExpatParser("books.xml");
$f1->ParserSetOption(XML_OPTION_CASE_FOLDING, 0);
$f2 = new FilterBuildBooks();
$f3 = new FilterNull();

$f2->SetListener($f3);
$f1->SetListener($f2);
$f1->Parse();
$books = $f1->listener->GetBooks();

```

We can make two observations about this piece of code:

- ❑ A NULL filter was used at the end of the chain since we neither want any output nor want to store the XML. Once parsed this design is better than making the `FilterBuildBooks` object a `FilterNull` because with this schema we can do something with the parsed XML document besides creating the objects. All our filters should pass events to a listener and we can use a NULL filter at the end.
- ❑ The books are obtained calling the `getBooks()` method of the `FilterBuildBooks` object through the parser; because the `FilterBuildBooks` is a member (listener) of the parser. If we try to obtain the books using `$f2->getBooks()` we will get an empty array.

The Flyweight Pattern

There's an interesting application of the **Flyweight** pattern in this example. Imagine we are processing a very large XML document where we create objects from the document. Many objects created from the document may have the same content for some of their properties if the elements repeat in the XML document. For instance, in our example, we may have many books from the same author.

In such cases we can use the Flyweight pattern to use an object that resolves the properties of the book objects.

Instead of storing the value of each property, we ask a `FlyweightFactory` object to return an object representing the property. We won't know if many objects share this object and the functionality of the object can be the same. The factorizing object can use a hashtable, or something similar, to store and retrieve properties reusing the same value for as many objects as it can, thus reducing the amount of memory consumed by the objects.

The flyweight pattern is an optimization pattern. It can be used when we are creating a very large number of objects and those objects may have properties with a long value that may be the same value for a big number of objects.

We can read about this pattern in the article at <http://www.xml.com/pub/a/2000/01/19/feature/index.html>.

Querying XML

If our data model is based on XML we'll be querying documents very often. Some examples of the need to query XML are:

- ❑ We have an XML file describing products and we want to find a price
- ❑ We have an XML file with a shopping cart and we want to calculate the total price
- ❑ We have an XML file with a log and we want to find some statistic
- ❑ We have an XML file for cars and we want to find cars below 20,000 USD, and with three doors

Querying an XML document means that we should traverse the XML document looking for particular information; what we do with the information once it is found is beyond the scope of the problem.

Example

Here we'll present a simple example to check how to solve the same problem using different approaches. In this case our example will show an XML file describing cars. This is the XML document:

```
<cars>
  <car>
    <type>F1</type>
    <brand>Ferrari</brand>
    <model>X1</model>
    <maxspeed>305</maxspeed>
    <hp>800</hp>
  </car>

  <car>
    <type>F1</type>
    <brand>Williams</brand>
    <model>W89</model>
    <maxspeed>298</maxspeed>
    <hp>1000</hp>
  </car>
```

```

<car>
  <type>F1</type>
  <brand>Brabham</brand>
  <model>B01</model>
  <maxspeed>311</maxspeed>
  <hp>1020</hp>
</car>
</cars>

```

And the query will be: "Find brand and model of F1 cars that can run at more than 300 kmh".

We have the following options to query an XML document:

- ☐ DOM
- ☐ XPath
- ☐ XSLT
- ☐ SAX

Using DOM To Query an XML Document

Using DOM to query an XML document is rather weird. Once we parse the document we obtain the DOM tree. Then we'll have to parse the tree to perform the query, and we don't have any useful querying feature in the DOM objects that may lead to such a strategy. We won't say that we can't use DOM to query a document, but we'll say that such a solution is uncommon.

If we do decide to use DOM for our query we can use the following code:

```

<?php
function GetContent($domnode)
{
    $content = '';
    foreach ($domnode->child_nodes() as $child) {
        if ($child->node_type() == XML_TEXT_NODE) {
            $content .= $child->content;
        }
    }
    return $content;
}

//$doc = xmldocfile("cars.xml");
$doc = domxml_open_file("cars.xml");

$root = $doc->document_element();
foreach ($root->child_nodes() as $element) {
    if ($element->node_type() == XML_ELEMENT_NODE and
        $element->tagname() == "car") {
        $cond1 = false;
        $cond2 = false;
        foreach ($element->child_nodes() as $car_element) {

```

```

        if ($car_element->node_type() == XML_ELEMENT_NODE and
            $car_element->tagname() == "type") {
            if (($type = getContent($car_element)) == "F1") {
                $cond1 = true;
            }
        }
        if ($car_element->node_type() == XML_ELEMENT_NODE and
            $car_element->tagname() == "maxspeed") {
            if (($maxspeed = getContent($car_element))>300) {
                $cond2 = true;
            }
        }
        if ($car_element->node_type() == XML_ELEMENT_NODE and
            $car_element->tagname() == "brand") {
            $brand = getContent($car_element);
        }
        if ($car_element->node_type() == XML_ELEMENT_NODE and
            $car_element->tagname() == "model") {
            $model = getContent($car_element);
        }
    }
    if ($cond1 && $cond2) {
        print ("Brand: $brand Model:$model<br />");
    }
}
?>

```

The code shows a typical approach to using DOM for querying. Traversing the DOM tree and finding the nodes that match some conditions is not that simple, and it isn't short either.

Using XPath To Query a Document

We haven't considered XPath so far in the chapter because XPath is a standard that is only useful to query an XML document and that's why it makes its stellar appearance in this section.

XPath is a standard to identify and retrieve portions of an XML document using location paths and predicates to filter the results. Many extremists say that XPath is the SQL of XML, but the truth is that many queries and computations can be solved easily using an XPath expression.

We have discussed XPath in detail in Chapter 7.

For our example we could have used the following XPath expression to solve the query:

```

/cars/car[normalize-space(maxspeed/text())>300 and normalize-
space(type/text())='F1']

```

The expression will return all the cars that match the criteria. We can then use the following PHP code to execute the XPath query, and extract the brand and model from the result:

```
<?php
// $doc = xmldocfile("cars.xml");
$doc = domxml_open_file("cars.xml");

$xmlpath = $doc->xpath_init();

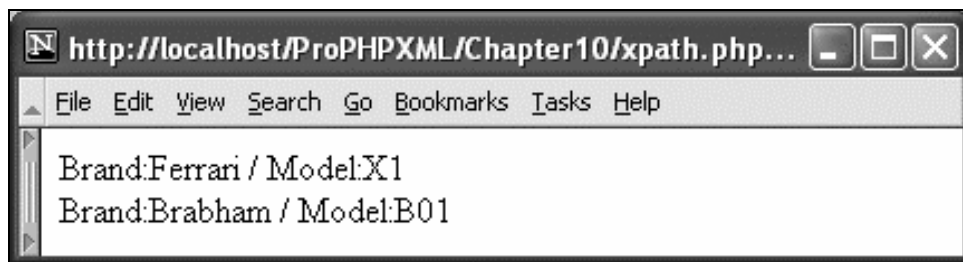
$ctx = $doc->xpath_new_context();

$result = $ctx->xpath_eval("/cars/car[normalize-space(maxspeed/text())>300
                        and normalize-space(type/text()='F1']]");

$nodes = $result->nodeset;

foreach ($nodes as $node) {
    foreach ($node->child_nodes() as $child) {
        if ($child->node_type() == XML_ELEMENT_NODE) {
            if ($child->tagname() == "brand") {
                print ("Brand:");
                foreach ($child->child_nodes() as $sub) {
                    if ($sub->node_type() == XML_TEXT_NODE) {
                        print ($sub->content);
                    }
                }
                print (" / ");
            }
            if ($child->tagname() == "model") {
                print ("Model:");
                foreach ($child->child_nodes() as $sub) {
                    if ($sub->node_type() == XML_TEXT_NODE) {
                        print ($sub->content);
                    }
                }
                print ("<br />");
            }
        }
    }
}
?>
```

The above code produces this simple output for our sampled XML document:



In PHP, the `xpath_eval()` function is part of the DOM extension, so using XPath implies parsing the XML document into a DOM tree. That's why the XPath function in PHP faces the same problems that we have found when using DOM in the previous example – it can be very resource-consuming or slow for larger documents.

If the document is small- to medium-sized then XPath is usually a very good alternative to query a document, since the query can usually be expressed with a simple expression minimizing the amount of code that we have to write for the query.

If the PHP XPath implementation is slow for large documents we can't blame XPath. It's easy to imagine that a SAX-based XPath processor will be faster than the DOM version, and we may also find an ad-hoc XPath processor, which will be even better.

At the time this book was written the only XPath implementation available for PHP was the `xpath_eval()` function in the DOM extension. It's logical to see new XPath processors emerging in the near future for PHP.

Here are some of the advantages and disadvantages of XPath:

Advantages	Disadvantages
A simple expression can solve many queries and computations.	Based on DOM, it inherits DOM's disadvantages – slow for large documents and uses a lot of resources. Thus, it can be unfeasible for huge documents.
Very easy to use.	Some complex queries or computations may not be solved with XPath.
Solid standard by the W3C.	

Using XSLT To Query a Document

XSLT is a very good tool to query a document. Whenever XPath is not enough to query a document we may think of using XSLT.

The same XSLT that is normally used to transform a document can be used to query it, obtaining a result instead of a transformation.

A query can be seen as a transformation where the document is transformed in the query result.

When using XSLT we can isolate each query into an XSLT stylesheet and write composite queries using `<xsl:include>`.

For example, the following XSLT stylesheet solves the query of our example:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="text" />

  <xsl:template match="/cars/car">
    <xsl:if test="normalize-space(maxspeed/text())>300 and
```

```

        normalize-space(type/text())='F1' ">
    Brand: <xsl:value-of select="brand" />
    Model: <xsl:value-of select="model" />
    <br />
</xsl:if>
</xsl:template>
</xsl:stylesheet>

```

The output after processing the XML file with the above XSLT would have been the same as the PHP script using XPath.

Here are the advantages of XSLT:

Advantages	Disadvantages
More powerful than XPath, can perform very complex queries or computations	Can be slow for huge documents
Fast enough for online processing of small- to mid-size documents	

Using SAX To Query a Document

When dealing with huge documents SAX is the only way out. The approach used to query a document is the same as when we transform or modify the document using SAX filters. We can write a filter for each query, and a combination of filters can be used for complex queries.

This SAX filter can be used to perform the query of our example:

```

class FilterQueryCars extends AbstractFilter
{
    var $results = Array();
    var $oneCar = Array();
    var $prop;

    function GetResults()
    {
        return $this->results;
    }

    function StartElementHandler($name, $attribs)
    {
        $this->prop = '';
        if ($name <> "cars" && $name <> "car") {
            $this->prop = $name;
            $this->oneCar[$this->prop] = '';
        }
    }

    function EndElementHandler($name)

```

```

    {
        if ($name == "car") {
            if (((trim($this->oneCar["type"])) == "F1") &&
                (trim($this->oneCar["maxspeed"])>300)) {
                $this->results[] = $this->oneCar;
            }
            $this->oneCar = Array();
        }
    }

    function CharacterDataHandler($data)
    {
        if ($this->prop) {
            if (!empty($data)) {
                $this->oneCar[$this->prop] .= $data;
            }
        }
    }
}

```

Note that the filter doesn't propagate events to a listener, so this filter has to be used at the end of the chain; query filters are usually placed at the end of the chain. For complex queries we may want to try to build the query using more than one filter. Chaining them reduces the complexity of each filter, and makes it easy to maintain them as well.

Here are some of the advantages and disadvantages of using SAX:

Advantages	Disadvantages
Scales very well for huge documents	No querying features, everything must be hand-made
Resource consumption is constant for documents of any size	Some complex queries may require a lot of code
It is fast	
Allows streaming	

Other Querying Alternatives

There are other querying alternatives for XML such as XQL and XQuery. We didn't cover them in this chapter since there isn't a PHP implementation of these alternatives. As they evolve and become stable we may find PHP extensions that support them.

Chapter 14 has a short introduction to XQuery. The specification is available at <http://www.w3.org/TR/xquery/>.

Abstraction Again

Querying an XML document is the problem where more solutions can be applied – XQL, XQuery, XSLT, XPath, DOM, SAX, and others can be used to query a document. So it's useful to have a query class abstracting queries. Also, it is almost always good to separate what the application needs from how it is done.

Caching Strikes Back

As with transformations, we might cache query results to reuse the result if the same query is applied to an unchanged document. The same considerations we made about caching transformations apply here.

Choosing a Querying Strategy

With so many options, choosing a querying strategy can sometimes be hard. We recommend the following strategy:

- ❑ If the documents are really huge then use SAX.
- ❑ If there are medium-sized documents then try using XPath. If performance is not satisfactory, or the query can't be solved, use XSLT.

Writing XML

Writing XML is a problem where an XML file has to be generated from a non-XML source (or it will be a transformation). For example, we may have some data in a database, a file, or objects, and we have to write an XML document. What we do with the XML document later is beyond the scope of the problem. Following are some instances of this problem:

- ❑ We have data in a relational database and want to produce an XML representation of the data
- ❑ We have data in a plaintext file that describes news and want to produce an XML RSS file
- ❑ We have some PHP objects and want to produce an XML file describing object properties
- ❑ We have a file in a proprietary format and want to convert it to XML for processing

We won't cover in this problem parsing the non-XML data, which is usually harder than writing the XML document.

Example

In this example we have the following text file describing a to-do list for some people:

```
Jim Smith, Go to the library and pick up books
Sidney, Make 13 copies of memo no:19
Kelly, Distribute memo no:5 by email
Sidney, Arrange meeting with foo managers
Kelly, Write a draft of the technical document for project X
Jim Smith, Distribute books to team members
Sidney, Record book loans in HR software
```

Our task is to generate a representation such as the following:

```
<to-do>
  <person>
    <name>Jim Smith</name>
    <tasks>
      <task>Go to the library and pick up books</task>
      <task>Distribute books to team members</task>
    </tasks>
  </person>

  <person>
    <name>Sidney</name>
    <tasks>
      <task>Make 13 copies of memo no:19</task>
      <task>Arrange meeting with foo&foo managers</task>
      <task>Record book loans in HR software</task>
    </tasks>
  </person>

  <person>
    <name>Kelly</name>
    <tasks>
      <task>Distribute memo no:5 by email</task>
      <task>Write a draft of the technical document for project X</task>
    </tasks>
  </person>
</to-do>
```

We can use the following tools to write XML documents:

- ☐ Manual writing
- ☐ DOM
- ☐ SAX

Manual Writing

Manual writing is a simple and intuitive way of writing XML documents – create a string with the desired XML. Simple PHP string management is used to generate the XML file. Generating the XML manually implies that the writing function must generate well-formed XML data. We will have to parse the strings and convert special characters such as & to the proper XML entities that represent them.

For our example we'll be using the following class that parses the .txt file with the to-do list to generate an array:

```
class ParseToDoTxt
{
    var $persons = Array();

    function ParseFile($filename)
    {
        $f = fopen($filename, "r");
```

```

        if (!$f) {
            return 0;
        }
        $txt = fread($f, filesize($filename));
        return $this->ParseString($txt);
    }

    function GetPersons()
    {
        return $this->persons;
    }

    function ParseString($txt)
    {
        $lines = Array();
        $line = Array();
        $lines = explode("\n", $txt);
        foreach ($lines as $line) {
            $line = chop($line);
            $line = explode(",", $line);
            if (!isset($this->persons[$line[0]])) {
                $this->persons[$line[0]] = Array();
            }
            $this->persons[$line[0]][] = $line[1];
        }
        return 1;
    }
}

```

The following bit of code generates the XML file:

```

$parser = new ParseToDoTxt();
$parser->ParseFile("todo.txt");
$persons = $parser->GetPersons();
$xml = '<?xml version="1.0"?>';
$xml .= '<to-do>';
foreach ($persons as $person => $tasks) {
    $xml .= '<person>';
    $xml .= '<name>'.XmlEntities($person).'</name>';
    $xml .= '<tasks>';
    foreach ($tasks as $task) {
        $xml .= '<task>';
        $xml .= XmlEntities($task);
        $xml .= '</task>';
    }
    $xml .= '</tasks>';
    $xml .= '</person>';
}

$xml .= '</to-do>';
print ($xml);

```

The `XmlEntities()` function replaces special characters with their corresponding XML entities:

```
function XmlEntities($data)
{
    $position = 0;
    $length = strlen($data);
    $escapeddata = "";
    for(;$position<$length;) {
        $character = substr($data, $position, 1);
        $code = Ord($character);
        switch($code) {
            case 34:
                $character = "&quot;";
                break;

            case 38:
                $character = "&amp;";
                break;

            case 39:
                $character = "&apos;";
                break;

            case 60:
                $character = "&lt;";
                break;

            case 62:
                $character = "&gt;";
                break;

            default:
                if ($code<32)
                    $character = ("&#" . strval($code) . ";");
                break;
        }
        $escapeddata .= $character;
        $position++;
    }
    return $escapeddata;
}
```

We can use this code as a method of the parsing class, or build a new `XmlWriter` class that uses the parsing class and constructs an XML string storing it as a property of the object. Without worrying about where we use the code we can see that the code is easy to understand and fairly easy to write too.

Manual writing is a simple and flexible way to write XML, however, such documents are difficult to maintain as encoding entities and producing well-formed XML documents is 100% the responsibility of the program.

Using DOM To Write XML Data

The DOM extension is usually used to parse an XML document to the DOM representation, but it can also be used to begin an empty document and add children, sub-children, and sub-sub-children until an XML document is constructed. The `dump_mem()` method then allows us to recover the XML document from the DOM tree.

The methods used to insert elements and attributes to the DOM tree are very useful when constructing XML documents.

This is the same code we used in the manual writing approach, but here we use DOM calls to generate the XML file:

```
$parser = new ParseToDoTxt();
$parser->ParseFile("todo.txt");
$persons = $parser->GetPersons();

// $doc = domxml_new_doc("1.0");
$doc = new_xmlDoc("1.0");

$node = $doc->create_element("to-do");
$root = $doc->add_child($node);

foreach ($persons as $person => $tasks) {
    $node_person = $doc->create_element("person");

    $node_name = $doc->create_element("name");
    $node_name_text = $doc->create_text_node($person);
    $node_name->add_child($node_name_text);

    $node_tasks = $doc->create_element("tasks");

    foreach ($tasks as $task) {
        $node_task = $doc->create_element("task");
        $node_task_text = $doc->create_text_node($task);
        $node_task->add_child($node_task_text);
        $node_tasks->add_child($node_task);
    }

    $node_person->add_child($node_name);
    $node_person->add_child($node_tasks);

    $root->add_child($node_person);
}

$xml = $doc->dump_mem();
print ($xml);
```

Using DOM we may skip calling the `XmlEntities()` function, since the DOM class is supposed to construct valid well-formed XML documents from our text.

In many ways using DOM is the best way to generate an XML file. The only problem we may face is that the PHP DOM extension is not very stable and changes to the extension may require changing programs that use it if we want to update our PHP installation.

Here are some of the advantages and disadvantages of using DOM:

Advantages	Disadvantages
Entities are generated by the DOM extension	Sometimes just manual writing is easier
Methods to ease the creation of complex XML documents	Programs depend on the unstable DOM extension of PHP
Code is easier to maintain	

Using SAX To Write XML Data

SAX can be used to generate XML documents. This is a use of SAX that is not seen very often and can lead to some good ideas once understood.

The idea is to write a SAX parser for non-XML data. We will use the examples of SAX filters we've already seen, write a new parser, and then apply all the filters we want to the new parser.

This is a parser that extends the `AbstractSAXParser` class, but parses a to-do plaintext file using the parsing class we used before:

```
class ToDoTxtParser extends AbstractSAXParser
{
    var $persons = Array();

    function ToDoTxtParser($txt)
    {
        $parser = new ParseToDoTxt();
        $parser->ParseFile($txt);
        $this->persons = $parser->GetPersons();
    }

    function Parse()
    {
        $this->StartElementHandler($this, "to-do", Array());
        foreach ($this->persons as $person => $tasks) {
            $this->StartElementHandler($this, "person", Array());
            $this->StartElementHandler($this, "name", Array());
            $this->CharacterDataHandler($this, XmlEntities($person));
            $this->EndElementHandler($this, "name");
            $this->StartElementHandler($this, "tasks", Array());

            foreach ($tasks as $task) {
                $this->StartElementHandler($this, "task", Array());
                $this->CharacterDataHandler($this, XmlEntities($task));
                $this->EndElementHandler($this, "task");
            }
            $this->EndElementHandler($this, "tasks");
            $this->EndElementHandler($this, "person");
        }
        $this->EndElementHandler($this, "to-do");
    }
}
```

As we can see, we parse the plaintext file and then traverse the array generating events for the XML elements that we want to produce. As we did with the manual writing approach, special characters must be converted to XML entities.

It would have been more efficient to make this class parse the plaintext file, but since we had little space we decided to reuse the parsing class we had. If we are parsing large non-XML files we should try to parse the file on-the-fly, that is, generate the proper SAX events as we read the non-XML file.

This is the PHP code used to generate an XML representation of the to-do plaintext file:

```
$f1 = new ToDoTxtParser("todo.txt");
$f2 = new FilterOutput();
$f1->SetListener($f2);
$f1->Parse();
```

Here are some of the advantages and disadvantages of using SAX:

Advantages	Disadvantages
Can be used as the start of a filter chain for many processing tasks	The parser is responsible for generating well-formed XML data converting special characters to entities
Allows streaming – we can generate the XML file as we read the non-XML data	

Summary

In most cases, the DOM approach uses more resources and is the slowest one too. DOM is a good solution to modify and write small- to medium-sized documents from non-XML data. DOM is not recommended for querying and transforming documents.

XSLT is the most interesting solution for modifying, transforming, or querying documents, unless the documents are so big that XSLT can't process them and we have to use SAX instead. Our XSLT stylesheets can perform complex queries or transformations with just a few lines of XSLT code, so if the size of the documents allows us to use XSLT, then it should surely be one of our primary weapons.

SAX is the only solution for processing really large files and it is also a good solution for a fast implementation of other processing problems. SAX filters and filter chains are the tools we should be constructing when SAX is our selected tool for a problem.

XPath, as we've seen in this chapter and in Chapter 7, is a good solution for not very complex queries since it requires just an XPath expression and a few lines of code. Since the XPath implementation of PHP is part of the DOM extension, it is not recommended for big files. SAX-based XPath implementations can be used to query large documents.

In this chapter we have seen examples that provide a good base, examples that use DOM, SAX, and XSLT for XML processing, and seen how to use these standards from PHP. After reading and practicing these examples we'll be able to select the best approach to a given problem and that's a key skill for any programmer dealing with XML data in the enormous ocean of derivatives available.

