Development of an Online Catalog system for an Autonomous Guided Vehicle using XML and Java

A Thesis submitted to the Division of
Graduate Studies and Advanced Research
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE

In the Department of Mechanical, Industrial and Nuclear Engineering
Of the College of Engineering
2001

by

Rahul Gurudatt Dhareshwar

B.E. (Production Engineering), University of Bombay
Bombay, India.
1999

Thesis Advisor and Committee Chair: Dr. Ernest L. Hall

Table of Contents

List of Figures

Abstract

This research attempts to provide more insight into the applications of XML for developing information catalogs. The Center for Robotics Research is dedicated towards developing a world-class autonomous unmanned vehicle, Bearcat III. Documentation of existing information for future use has always been an issue. There have been previous attempts at developing an online interactive system that would help existing team members get a better understanding of Bearcat III.

The information is stored in a validated and well-formed XML file. In order to keep this work focused, the highlight was made on critical information that would be significant to the robotic team members in terms of the approaching International Ground Robotics competition at Oakland University. An XML file consisting of sponsors as well as critical parts of the vehicle was developed. The list was made comprehensive to include all parts that make up Bearcat III. This self-describing document is then made available to the outside world using a simple web browser that is XML enabled. A Document Object Model (DOM) was used to provide a means for working with information in the XML document. Also a Java front-end interface was developed that would parse through the XML files using SAX (Simple API for XML) and an XML parser. Java being a portable programming language and XML being a portable document format, would allow this information to be accessed independent of the platform. With this synergy in mind, let's look at how these two technologies fit together, both today and tomorrow.

Rahul:

Please rewrite your abstract answering these four questions in order:

1. What you have done?
2. How you have done it?
3. What results you have achieved?
4. What is the significance of the results?

You should write an abstract to be informative not play a guessing game.

Acknowledgements

I am grateful to my advisor Paul E. Geier Professor of Robotics, Dr Ernest Hall for giving me an opportunity to work in the exciting field of XML and Java. He has always been a source of inspiration for me, and provided the support and encouragement whenever I needed it. I would like to personally thank him for all his assistance and support throughout my graduate studies at the University of Cincinnati.

I wish to thank the members of my masters defense committee, Dr. Shell and Dr. Huston for their valuable suggestions and support on this research. Their comment on this research is highly appreciated, and has helped me focus myself better on the topic.

I would like to take this opportunity to thanks the entire faculty and staffs at the MINE department and the Information Systems department at the University of Cincinnati, for helping me in pursue my Masters program.

I also gratefully acknowledge the support and assistance of all undergraduate and graduate students of the Center of Robotics Research. I would also like to thank Satish Natarajan, and Jaigainesh for their valuable insights on this topic.

Most importantly I extend my gratitude to my parents who have always encouraged me throughout my education and career. They stand behind all my success.

I think you have a well written document but it still needs looked over carefully.
My biggest concern is your results. You have some but they are not tested. Do you have a complete data base of sponsors or just a partial list. I have the same concern for the parts list. Is it complete? How does someone update the data bases you have? You sem to have the GUI part OK but it is only useful if you have good data.

1. Introduction

The Center for Robotics Research at the University of Cincinnati has been dedicated towards developing a world-class autonomous unmanned vehicle, Bearcat II. Bearcat II has been an active participant at the International Ground Robotics competition, and secured the third position at IGRC 2000. The vehicle has been built by the indigenous ideas and efforts of the previous team members, and modified by the existing team members to fit the contest regulations. The Bearcat has undergone two major versions, with the current version Bearcat III being more compact and rigid. Thus there has been considerable transfer of knowledge and ideas, and a need to document valuable data has always been stressed upon.

You should use an indent for consecutive paragraphs.

This world-class unmanned vehicle is the result of culmination of great ideas from both undergraduate and graduate students as well as generous sponsors, both corporate and individuals. Every year in the spring quarter, the team is faced with the problem of attracting more sponsors, so that finances could be distributed in developing applications that use cutting edge technology. The current approach towards this has been very haphazard and has resulted in the team spending more time and effort in developing a sponsor list, that developing the robot.

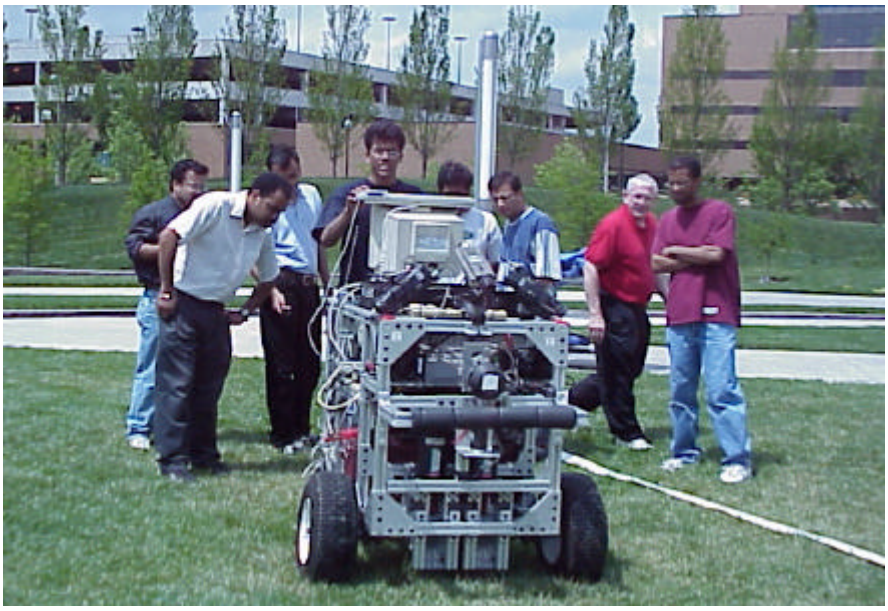


Figure No. 1 Bearcat III robot [8]

1.1 Objective

Associated with this, is a goal to develop an approach towards management, display and organization of data, one that is open and accessible to all, and delivers a broad spectrum of capabilities. This requires use of a methodology, that is not myopic to meet only current needs, but also extensible to meet future requirements and adaptable enough to incorporate emerging technologies and requirements. The Extensible Markup Language (XML) developed by the World Wide Web Consortium (W3C) [13] provides a technology concerned with the description and structuring of data.

With this in mind, the focus was to develop a valid and well-formed XML document, which would be self-describing. This XML document would provide for management and organization of data, which could be made available to the world using the World Wide Web or Application programs. The salient features are

- A comprehensive XML document that would capture information about previous, current and potential sponsors as well as parts critical to the robot.

- A Document Type Definition (DTD) that would describe the rules that the XML document is trying to follow. This would provide a well structure approach towards future management of information.
- Use of a web browser, which would allow for presentation of information. A Document Object Model (DOM) is implemented to work with the XML documents and provide required information to the user.
- Use of an XML parser that would validate the XML document.
- A user-friendly Java application program that would use the more efficient SAX (Simple API for XML) to analyze XML documents and extracts relevant information.

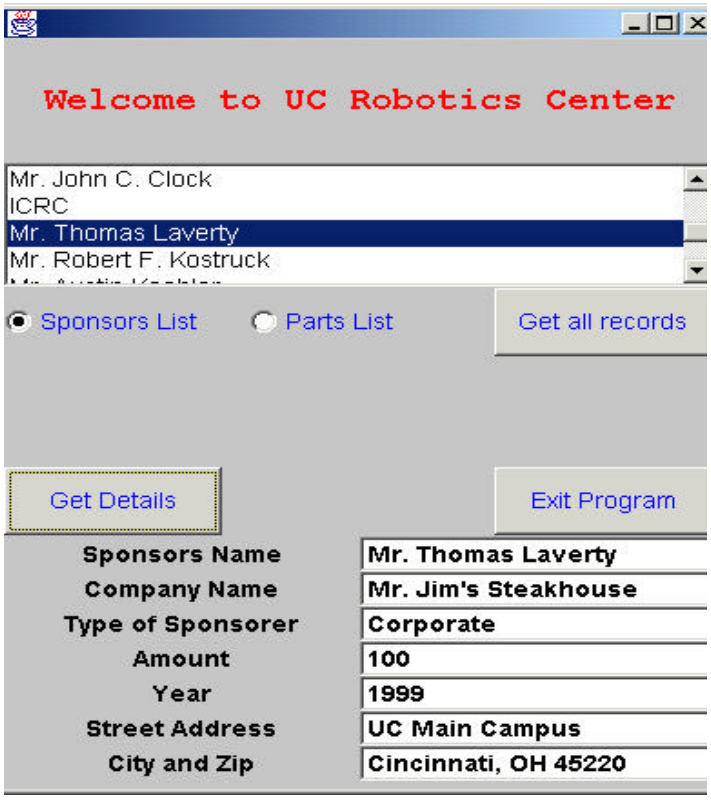


Figure No.2 GUI (front – end) of Java application

1.2 Organization.

This work has been organized into several chapters to document the ideas and technologies that I have used.

Chapter two focuses on the various components of the vehicle, the manner in which the subsystems interact with their environment, and their significance to the overall performance of the system.

Chapter three gives an introduction to the Markup Languages, their significance in terms of disseminating information across a neutral platform to a wide variety of users. The powerful technology of XML is touched upon, and its close association with Java is explored.

Chapter four deals with development of well-formed and valid XML documents.

Chapter five provides an outlook into analyzing XML documents using the DOM technology.

Chapter six explores the more efficient SAX interface, along with certain advantages limitations as compared to DOM.

Chapter seven gives a detailed look into the architecture of both, DOM and SAX implementations.

The final chapter provides conclusions drawn from the work, as well as recommendations for future work.

2. Components of Bearcat III

The purpose of this chapter is to explore the various functional components that make up such a sophisticated system, Bearcat III. The strategy used was to decompose the entire unit into simpler subsystems and then study each system and its significance to the performance of the entire vehicle. Bearcat III can be broadly categorized into the following subsystems:

- Mechanical system.
- Vision system.
- Power system.
- Navigation system.

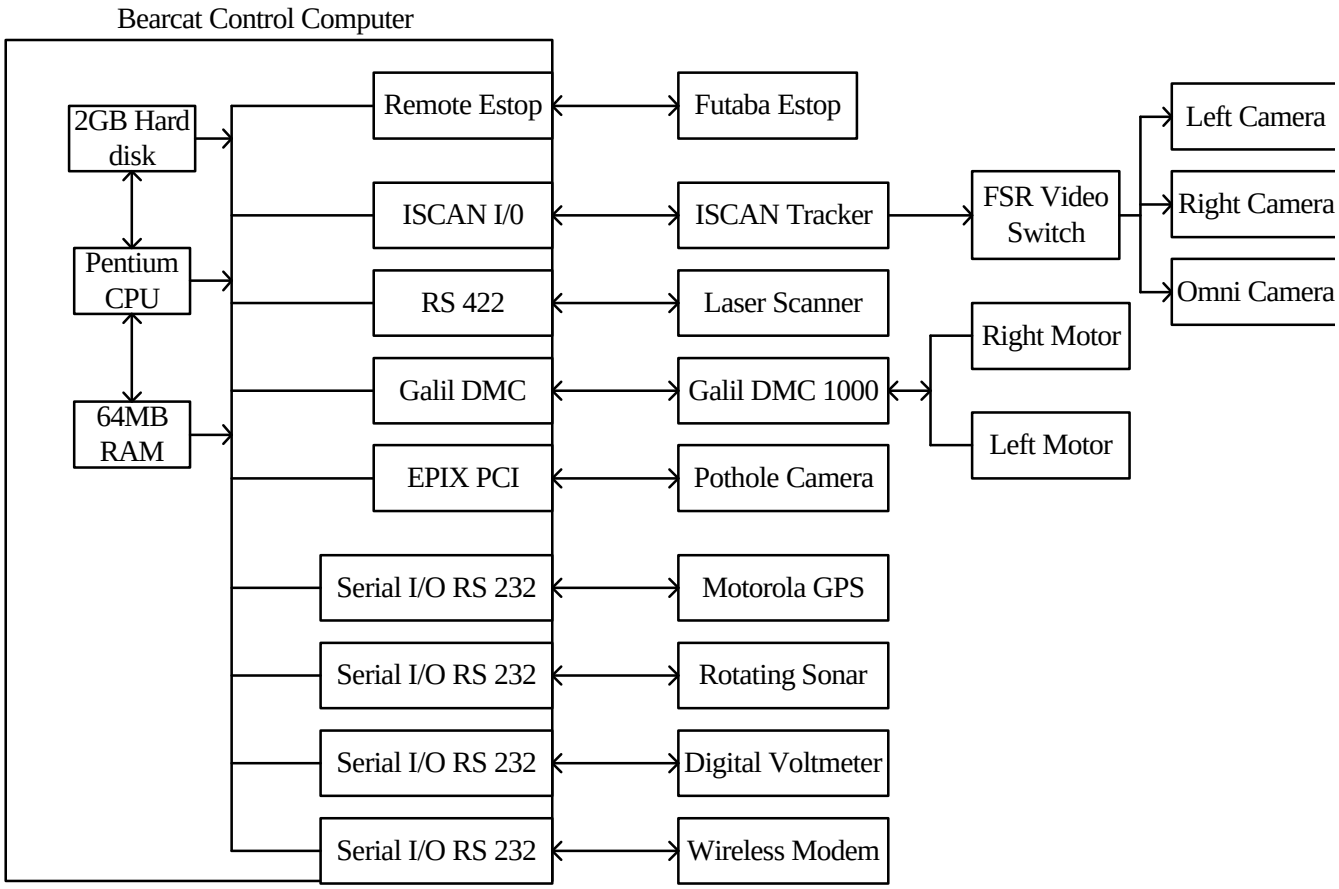


Figure No. 3 Block diagram for Bearcat III [8]

Our Bearcat III robot is designed for planning and reactive control using

- Three-dimensional line following.
- Obstacle detection with sonar.
- Obstacle detection with laser scanner.

2.1 Mechanical system.

This system is designed taking into account the conditions and payload requirements of Bearcat III. The vehicle navigates using two 36 volts 15 amperes motors. The motors drive the front left and the right wheels separately through two independent gearboxes, which increase the motor torque by a factor of 40. This enables a zero turning radius by rotating one wheel in the forward direction and the other in the reverse direction. This unique design offers the ability to make a turn about the center of axis of the drive wheels thereby providing the vehicle exceptional maneuverability. The zero turning radius is achieved by using a rear castor wheel. The power to each motor is delivered from an AMC DC 48A amplifier that amplifies the signal from the Galil DMC Motion Controller.

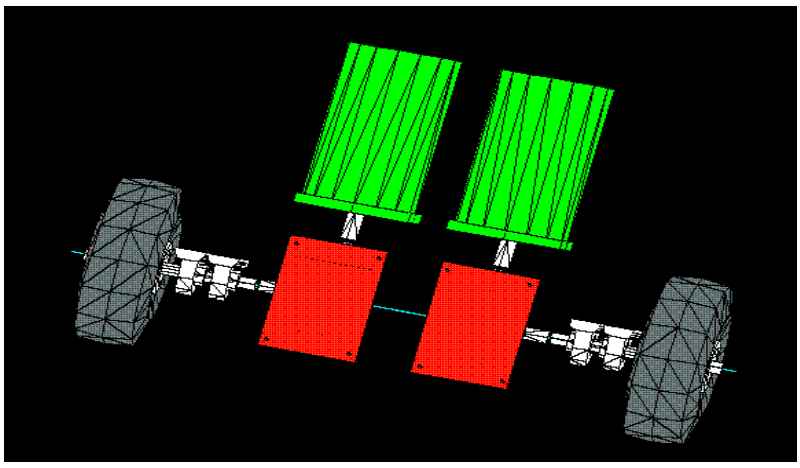


Figure No. 4 Model of the Power train [5]

2.2 Vision system.

The vision system is a critical component that assists the vehicle following a line, and avoiding potholes that it might encounter on its course. The vehicle has three cameras, two of which are used for line following and the third is used for pothole detection.

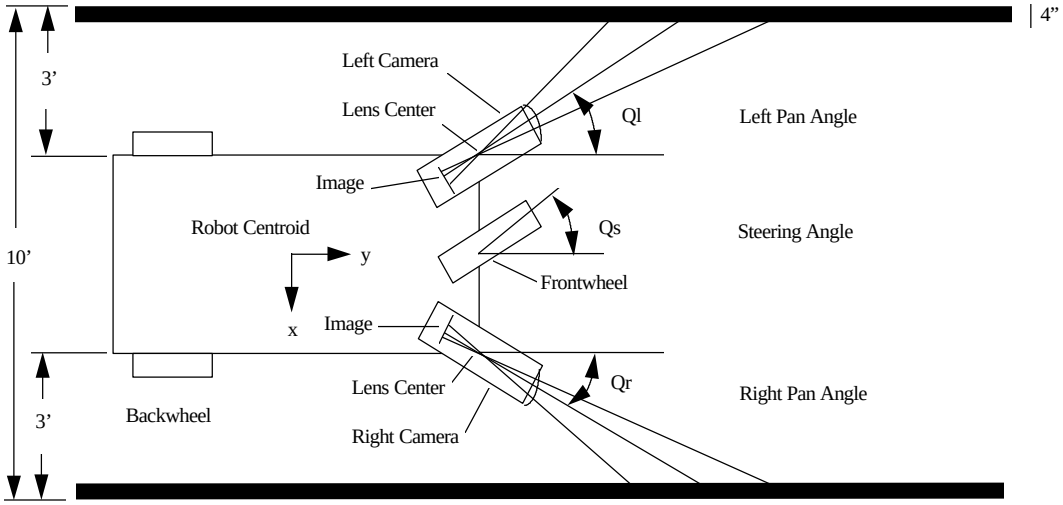


Figure No. 5 Overview of the Vision system [7]

Line following: The course that the robot follows is bounded using solid or dashed white lines. The vehicle uses two JVC CCD cameras to digitize the vision system from a 3-D coordinate system to a 2-D coordinate system. Image processing is done by an ISCAN tracking device. A CCSU-8BW video switch from FSR, Inc., alternates between the two cameras depending upon the visibility of the marker.

Pothole detection: As per the contest requirements, the vehicle is required to detect and avoid potholes in its course. The reaction distance of the vehicle is twelve feet. A monochrome Panasonic CCD camera is used to capture the course ahead of the robot.

The data from the camera is passed to a PIXCI SV4 imaging board, manufactured by Epix, Inc.,

2.3 Power system.

The vehicle draws power from three 12 volts Trojan marine batteries that are connected in series to provide the 36 volts required by the motors. In ideal conditions, the batteries have a run time of about 7 hours between charges, the charging time being 8 hours [6].

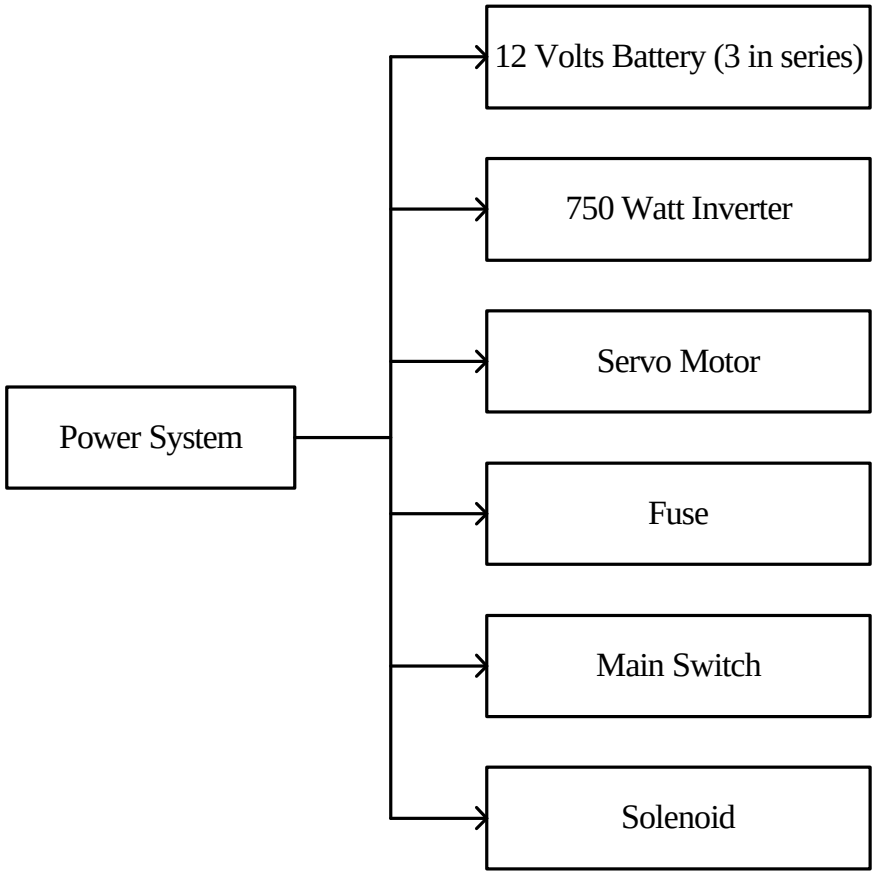


Figure No. 6 Elements of the Power system [9]

Maurice has put a new drawing o fthe power system on the robotics web page. This one makes little sense.

2.4 Navigation system.

Three components that assist in navigating the robot on its trajectory are:

- Sonar system.
- GPS.
- Laser Scanner.

Sonar system: The objective of this subsystem is to provide reliable measurements in terms of the distance of the vehicle from the obstacle, and the angle to the obstacle. It uses a Polaroid ultrasound ranging kit and RS 232 interface card.

Global Positioning system: The GPS is necessary due to modifications in the contest requirements. The GPS sets up a fixed base point and tracks the vehicle as it moves from one point to point another, updating the new base point with every pass.

Laser Scanner: The laser scanner would be used for avoiding any obstacles that the vehicle might encounter while navigating the course. The principle behind this device is reflection of laser light from the obstacle. The time of flight gives the vehicle an idea of how far the obstacle is, allowing for earlier reaction time. The system is composed of a SICK laser scanner and interfaces with the computer through a high speed RS 422 interface card.

3. Need for XML

The Internet has grown rapidly since its inception and is anticipated to expand rapidly over the next few years. The Internet and the World Wide Web represent significant advancements for retrieval and dissemination of information across a large contingent of users. One of the central, if accidental innovations of the World Wide Web was the advent of a platform independent graphical user interface markup language. The solution, in theory, is very simple, the use of tags to display and format the information.

3.1 Markup Languages: SGML and HTML

Standard Generalized Markup Language (SGML) is an international standard for the description of electronic marked-up text. Essential SGML is used for describing other markup languages – a metalanguage – and has since proved valuable in many large publication applications. By *markup language* we mean a set of markup conventions used together for encoding texts. A markup language must specify what markup is allowed, what markup is required, how markup is to be distinguished from text, and what the markup means [3]. SGML provides the means for doing the first three; documentation such as these guidelines is required for the last. Unfortunately, SGML is such a complicated language that it's not well suited for data interchange over the web.

HyperText Markup Language (HTML) is used for publishing hypertext on the World Wide Web. It is a non-proprietary format based on the more powerful SGML, and can be created and processed by a wide range of tools. Technically speaking, HTML is an

application of SGML. The idea, is to present any HTML document (or web page) in any application that is capable of interpreting the HTML (termed a web browser). HTML instructions divide the text of a document into blocks called *elements*. These can be divided into two broad categories -- those that define how the BODY of the document is to be displayed by the browser, and those that define information about the document, such as the title or relationships to other documents [1].

3.2 XML

Extensible Markup Language (XML) combines the useful features from both HTML and SGML. XML is actually a subset of SGML and is designed to fully compatible with SGML. Therefore, any valid and well-formed XML document automatically qualifies to be a valid SGML document, which is not the case with HTML documents. To put in simple terms, what HTML does for display, XML is designed to do for data interchange. The fundamental difference between the two:

- HTML is designed for a specific application, to convey information to user through a graphical user interface termed the web browser. HTML is concerned with presentation of the document.

- XML has no specific application, it application are only limited by the developer's imaginations. XML is concerned with describing the document and attempting to follow a set of rules laid down by a DTD.

Essentially, XML describes a syntax that you use to create your own languages. This is a true representation of the word Extensible, allowing us to shape the data in any way so that it is understood by a particular application. The benefits of XML have become more apparent since the creation of formats for interchange of information. There have already been numerous projects to describe industry specific standards for describing data. For example, Wireless Markup Language (WML) is a markup language based on XML, and is intended for use in specifying content and user interface for narrowband devices, including cellular phones and pagers.

3.3 XML Parsers

Parsers are programs, which are able to interpret XML syntax and extract the information out of it. The parsers are used within the application programs, keeping the application program independent of the XML document. A parser is commonly referred to as an XML processor.

At present two major Application Programming Interfaces (API) define how XML parsers work: DOM (Document Object Model) and SAX (Simple API for XML). The DOM specification defines a tree-based approach to navigating an XML document [2]. In other words, a DOM parser processes XML data and creates an object-oriented hierarchical representation of the document that you can navigate at run-time.

The SAX specification defines an event-based approach whereby parsers scan through XML data, calling handler functions whenever certain parts of the document (e.g., text nodes or processing instructions) are found [2].

Internet Explorer 5.0 ships with an XML parser and a library called MSXML, which includes a DOM implementation. These capabilities of IE 5.0 have been exploited in parsing through XML documents.

For the Java application program developed, I used the XP parser developed by James Clark. XP is an XML 1.0 parser written in Java. It is fully conforming: it detects all non well-formed XML documents. The reason I chose XP, among the various other available parsers is:

- XP is designed to be 100% conformant to the XML 1.0 specification.
- High performance. XP aims to be the fastest conformant parser in Java.
- Layered structure. In addition to a normal high-level parser API, XP provides a low-level API that supports the construction of different kinds of XML parser (such as incremental parsers).

The step-by-step instructions for installing the parser are given Appendix B.

3.4 Java and XML: A perfect match.

Both Java and XML are proven and powerful technologies. Java's modular and portable structure has made it the language of choice for writing distributed applications that run on every platform. This makes Java ideal for both internal distributed computing and business-to-business computing, because one Java application can easily speak to another over the Internet thanks to Java's built-in security [12]. The platform-independent

Java language lacks a way to exchange platform-and application-neutral data, which is where XML comes in.

Java and XML are two powerful technologies that together make a very strong team. They can run on any platform, which eliminates concerns about hardware, operating system, and network because Java and XML use standard Internet protocols. Java is, in fact the ideal counterpart for XML, and the reason can be summed in a single phrase: "Java is portable code, and XML is portable data."[4]

4. Well-Formed and Valid XML Documents

XML is designed to conform to the author's needs, allowing a greater degree of structural and stylish customization than that possible with HTML. Even though it is a subset of SGML, XML is designed to overcome the shortcomings of SGML documents when used over the web. To begin with, XML allows us to create our own self-describing tags, something which not possible in HTML. In order to maintain consistency over XML, each document must be well formed as well as valid.

4.1 Well Formed XML

Well-formed XML removes the restraints imposed by HTML when creating new documents. The creativity of an XML document is limited by the imagination of the developer. Well-formed documents do not have to be created in a structured environment, against a pre-defined set of structural rules, but merely have to comply with XML *well-formedness constraint*s. These constraints require that elements, which are named content containers, properly nest within each other and use other markup syntax correctly [2]. Well-formed XML elements are defined by their use, not by a rigid structural definition, allowing authors to create elements in response to their development. This flexibility offers authors greater control over document processing and design than in traditional SGML environments, in which structure must be formally defined in a Document Type Definition (DTD) before any documents can be written.

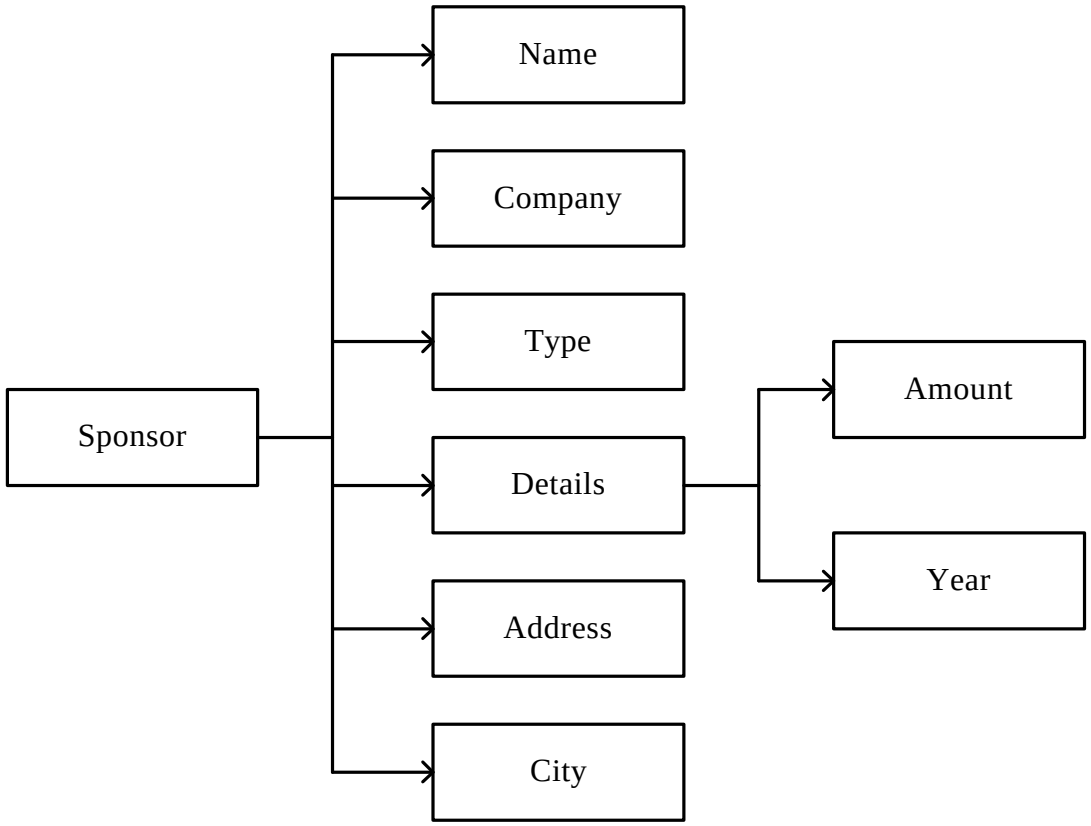
Elements: The words between < and > characters are XML tags. The information in our document is contained within the various tags that constitute the markup of the document. Tags are always paired together, unlike HTML where missing an end tag is not a serious

offence. All of the information from the start of a start – tag to the end of an end – tag constitutes an elements. So

- <Name> is a start tag.
- </Name> is an end tag.
- <Name>Rahul Dhareshwar</Name> is an element.

The text between the start and end tags is referred to as element content. For the sponsor's list (attached in Appendix) the whole document starting with <Sponsors> and ending with </Sponsors> is an element, which happens to include other elements. Such an element is called as the root element.

Here's the hierarchy that is used for the Sponsor list.



and the hierarchy for the Parts list

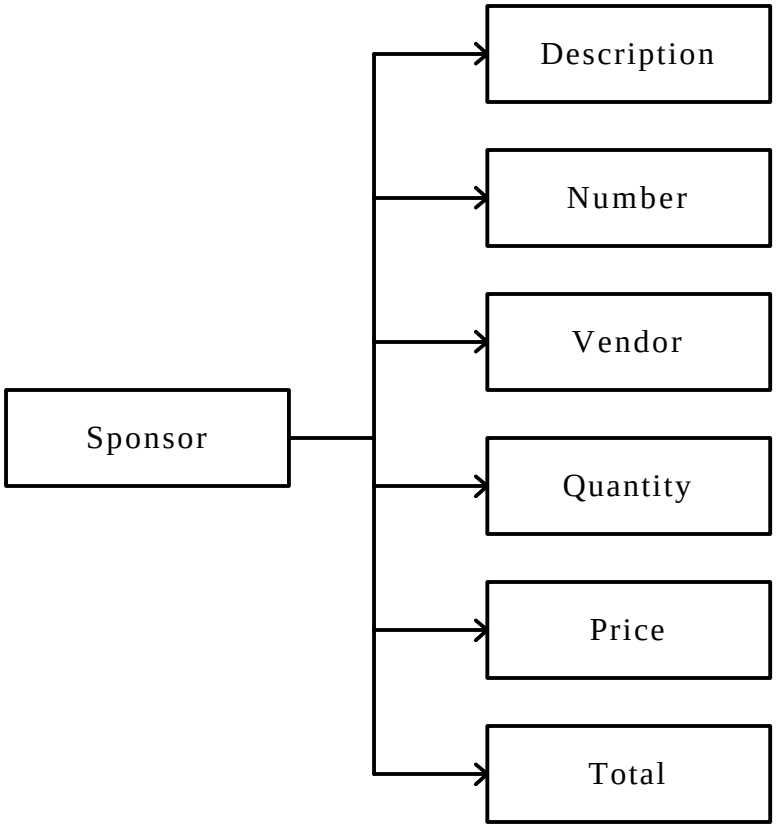


Figure No. 7 The XML structural approach

I created the following tags, which are short yet, descriptive enough for anyone looking at it to know what they represent. Of course, you may name your tags somewhat differently, but the trick is to keep things very clear and sensible

There are certain rules that the elements are supposed to follow. These rules are fundamental to the understanding of XML.

- Every start – tag must have a matching end – tag.
- Tags cannot overlap one another.
- XML documents can have only one root element. In our case, <Sponsors> is the root element for the Sponsor.XML document.
- Element names must obey XML naming conventions.

- XML is case – sensitive.
- XML will keep white space in your text.

All the above rules have been strictly adhered to in the XML document created for the Sponsors.

Attributes: They are simple name value pairs associated with an element. They are attached to the start tag as shown below, but not to the end tag.

<Sponsor type = 'Corporate'>

<Name>Kroger</Name> ……

</Sponsor>

It seems that attributes capture information that could be stored in an element otherwise. There have been many debates in the XML community about whether attributes are necessary. There is really no difference between what attributes provide in terms of functionality when compared to elements [13]. I chose to represent all information using elements, to provide a consistent approach.

XML Declaration: XML provides the XML declaration for us to label documents as being XML. Some additional information could also be given to the parsers through the declaration. The XML declaration used in the Sponsor list looks like this.

<?xml version='1.0' standalone='no' ?>

Even though it does not seem to make much sense, an XML document without a declaration is not well formed.

The main reason for documenting all these rules about writing a well-formed XML document is so that our parser (XP) can read in the data, and easily distinguish markup from information.

4.2 Valid XML

Valid XML is a more rigid or formal requirement that the document needs to follow as compared to its well-formed counterpart. Documents are well formed; otherwise they would not be XML documents. Valid documents must conform not only to the syntax, but also to the DTD (Document Type Definition). DTD is a set of rules that defines what tags appear in a XML document, so that viewers of an XML document know what all the tags mean [2]. DTDs also describe the structure of a document. Thus primary difference between valid and well-formed XML is their relationship to a *document type definition*. Well-formed XML is designed for use without a DTD, whereas valid XML explicitly requires it.

4.3 DTD

A DTD is a Document Type Definition. It establishes a set of constraints for an XML document, or a set of documents. DTD is not a specification on its own, but is defined as a part of the XML specification. Put in simple terms, the XML document tries to follow the rules laid down by the DTD. Being able to define such rules will become more important as we exchange, process and display XML in a wider environment, such as in business-to-business or e-commerce scenario [1].

Separating the XML data description from individual applications allows all cooperating applications to share a single description of the data, known as the XML vocabulary. A group of XML documents that share a common XML vocabulary is known as a document type, and each individual document that conforms to a document type is a document instance.

In addition to ensuring that XML data is simply well formed, most XML applications will also need to:

- Describe document structure, preferable in a rigorous and formal manner.
- Communicate the document structure to other applications.
- Check that the required elements are present.
- Check that no disallowed elements are included.
- Enforce element content, tree structure, and element attribute values.

In addition, DTD's can be either Internal or External. An Internal DTD is one that is included within the XML document. An External DTD on the other side is a set of declarations that are located in a separate document, using the .dtd filename extension. I have used an External DTD, since it easier from a maintenance standpoint. Also it keeps the metadata separated from the data.

The Document Type (DOCTYPE) Declaration is used to link the XML documents using the markup with the specific DTD. The DOCTYPE structure used in the Sponsor.xml file is as:

<!DOCTYPE Sponsorers PUBLIC file:///c:/XML/Sponsorers.dtd>

The above DOCTYPE declaration consists of

- The usual XML tag delimiters ("<" and ">")
- The exclamation mark ("!") that signifies a special XML declaration.
- The DOCTYPE keyword.
- The name of the document element. This in our case is the document element Sponsorers.

- Use of keyword, Public that allows a non-specific reference to the DTD via a Uniform Resource Identifier (URI).
- A location to associate the external DTD with the document.

4.4 Constructing our own DTD.

DTD declarations are delimited with the usual XML tag delimiters ("<" and ">"). Like the DOCTYPE declaration, all DTD declarations are indicated by the use of the exclamation mark ("!") followed by a keyword and specific parameters.

<!keyword parameter1 parameter 2 .. parameterN> [2]

There are four basic keywords used in DTD declarations:

Keyword	**Description**
ELEMENT	Declares an XML element type name and its permissible sub-elements.
ATTLIST	Declares XML element attribute names, plus permissible attribute values.
ENTITY	Declares special character references, text macros and other content.
NOTATION	Declares external non-XML content and external application.

Table No. 1 [2]

The first two keywords are essential for describing an XML document or data model. The latter two provide useful shortcuts for creating documents with reusable content, plus methods for handling non-XML data.

The DTD that I have created makes most use of Element type declarations, which are described below.

Element Type Declarations: Elements are the basic building blocks of XML data. A typical document would include several element types that would be perfectly nested within one another. A point to be noted here is that the only mandatory component of an XML document is an element – even the most trivial XML document would have at least the document element.

The element type declaration can have one of the two following forms:

<!ELEMENT name category > [2]

<!ELEMENT name (content_model) > [2]

The category and content_model parameters describe what kind of content (if any) may appear within the elements of the given name.

There are five categories of element content:

Content Category	**Description**
ANY	Element type may contain any well-formed XML.
EMPTY	Element type may not contain any text or child elements.
Element	Element type contains only child elements.
Mixed	Element type may contain text and/ or child elements.
PCDATA	Element type may contain text (character data) only.

Table No. 2 [2]

Content Models are used to describe the structure and content of a given element type. This may be:

- Character data (PCDATA content).
- One or more child element types (element – only content).

- A combination of the two (mixed content).

The following document excerpt shows an element with PCDATA content:

```
<Address>2900 Vine St, Apt F</Address>
```

Elements in the PCDATA content category only allow character data, and the #PCDATA is the only parameter allowed in such a content model. An example is below:

```
<! ELEMENT Address (#PCDATA) >
```

Element Content on the other hand, may contain only other child elements. An example document fragment that would conform to this category would be:

```
<Sponsor>
    <Name>Dr. Ernest L. Hall</Name>
    <Amount>100</Amount>
</Sponsor>
```

In this document, no text appears outside any of the tags. In this case, Sponsor would be declared in the DTD as having only element content, like so:

```
<! ELEMENT Sponsor (Name, Amount) >
```

The comma-separated list of child element names within the parentheses in a declaration is known as a sequence list. In the above example, only two child elements are allowed, and they must appear in the order shown in the declaration. However, all element content models may use combinations of sequence lists and choice lists. These are further explained below:

List Operator	Description
, (The comma)	Sequence – child elements must appear in the specified order.
| (vertical bar)	Choice – only one of several child elements are permitted.

Table No. 3 [2]

In the DTD that I have built, all elements use the sequence operator.

Element Models – Cardinality: Cardinality operators define how many child elements may appear in a content model. There are four cardinality operators:

Cardinality Operator	Description
[none]	The absence indicates that only one instance of child element is allowed.
?	Zero or one child element – optional singular element.
*	Zero or more child elements – optional element(s).
+	One or more child elements – required element(s).

Table No. 4 [2]

The following element type declaration used in the DTD conforms to the above syntax:

<! ELEMENT Sponsor (Name, Type, Details*, Email?, Address?, City?) >

The complete DTD for both the Sponsor and Parts list is given in Appendix C.

5. Working with XML documents using DOM.

Now that we have got our information in XML format, the next thing that we would like to do is, use the information in applications that we write. This could include accessing, changing as well adding data. The XML Document Object Model (DOM) is a programming interface that allows XML documents to be accessed and manipulated.

5.1 The need for DOM.

For example, we could take our <Sponsor> XML, which looks like this:

```
<Sponsor>
        <Name>Dr. Ernest L. Hall</Name>
        <Type>Individual</Type>
         <Details>
                <Amount>100</Amount>
                <Year>2001</Year>
        </Details>
        <Address>9256 Village Green Drive</Address>
        <City>Cincinnati, OH 45242</City>
 </Sponsor>
```

and structure it as an object model such as the following:

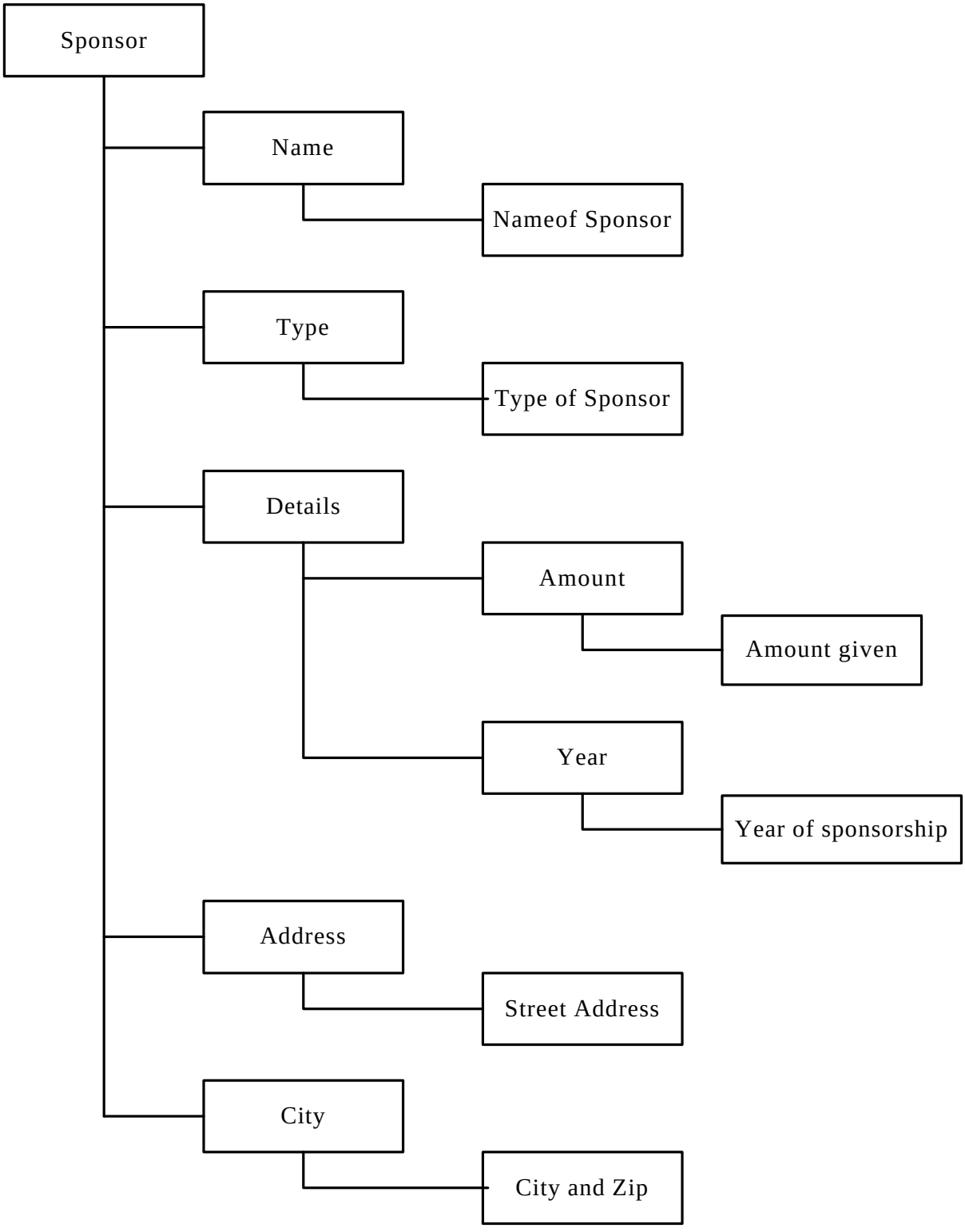


Figure No. 8 Object Model Structure

Some of the elements represent objects (shown in bold) while the rest are properties for that object. If we were writing code to deal with a Sponsor, this object model would be easier to process that information, and would probably include methods to provide some functionality. We need an object model that can model any XML document regardless of how it is structured. The Document Object Model (DOM) takes this generic approach. The DOM usually sits on top of the XML parser layer and provides the information to the specific application. A representation would be as below:

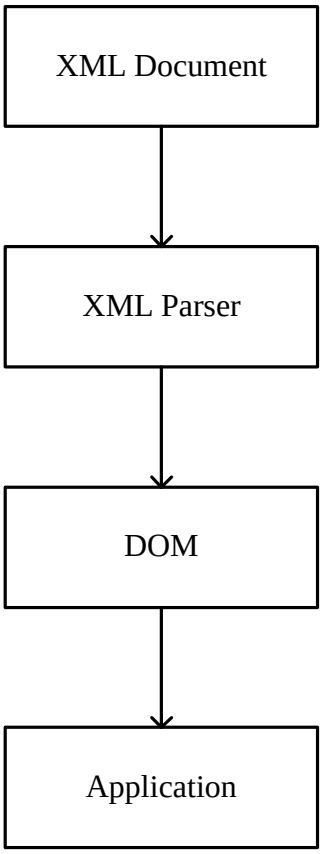


Figure No. 9 DOM and XML Parser

5.2 Using the DOM interfaces

The DOM usually deals with interfaces. Interfaces are a collection of methods and properties that one or more objects may support [4]. To get an idea of what interfaces are involved in DOM, lets take a look at very simple example from our XML document:

```
<Sponsor>
        <Name>Name of the Sponsorer</Name>
</Sponsor>
```

It would be represented in DOM by:

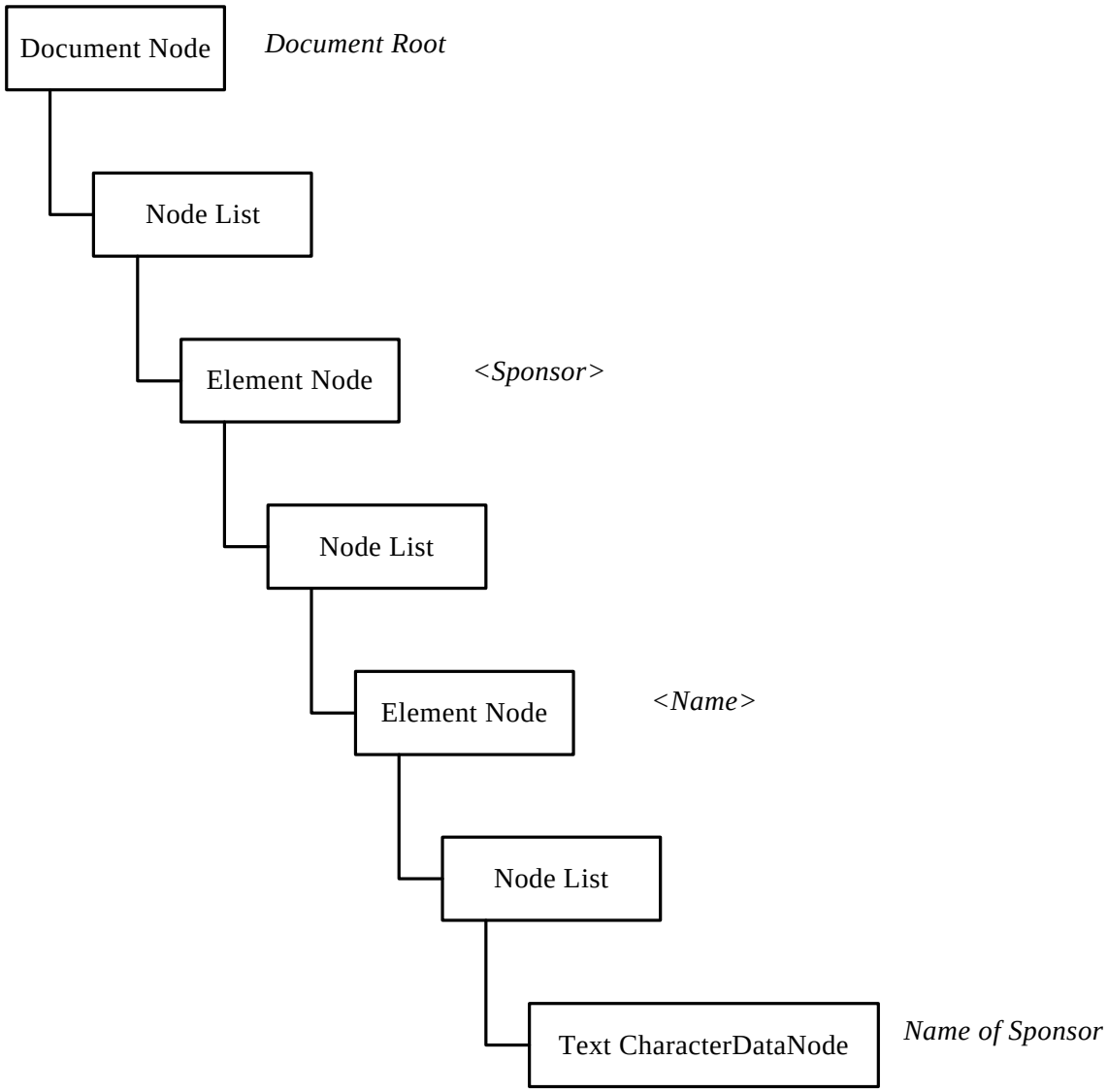


Figure No. 10 DOM equivalent of XML

Each box represents an object that shall be created; the names in the boxes are interfaces that shall be implemented by each object.

We shall be using some of the Internet Explorer 5 capabilities to display XML documents. Internet Explorer 5 ships with a library called **MSXML** that includes a DOM implementation. The Microsoft XML parser is a COM component that comes with Microsoft Internet Explorer 5.0. Once you have installed Internet Explorer 5.0, the parser is available to scripts inside HTML documents and ASP files. The Microsoft XMLDOM parser features a programming model that:

- Supports JavaScript, VBScript, Perl, VB, Java, C++ and more
- Supports W3C XML 1.0 and XML DOM
- Supports DTD and validation

We shall be using JavaScript on a web page using Internet Explorer 5.0. JavaScript is a scripting language and differs from Java, which is a pure programming language. Scripting languages combine tools from programming languages to make them more concise and usable [5]. JavaScript is a scripting language designed to extend the functionality of web pages.

5.3 DOM Core

The DOM Core provides the following interfaces:

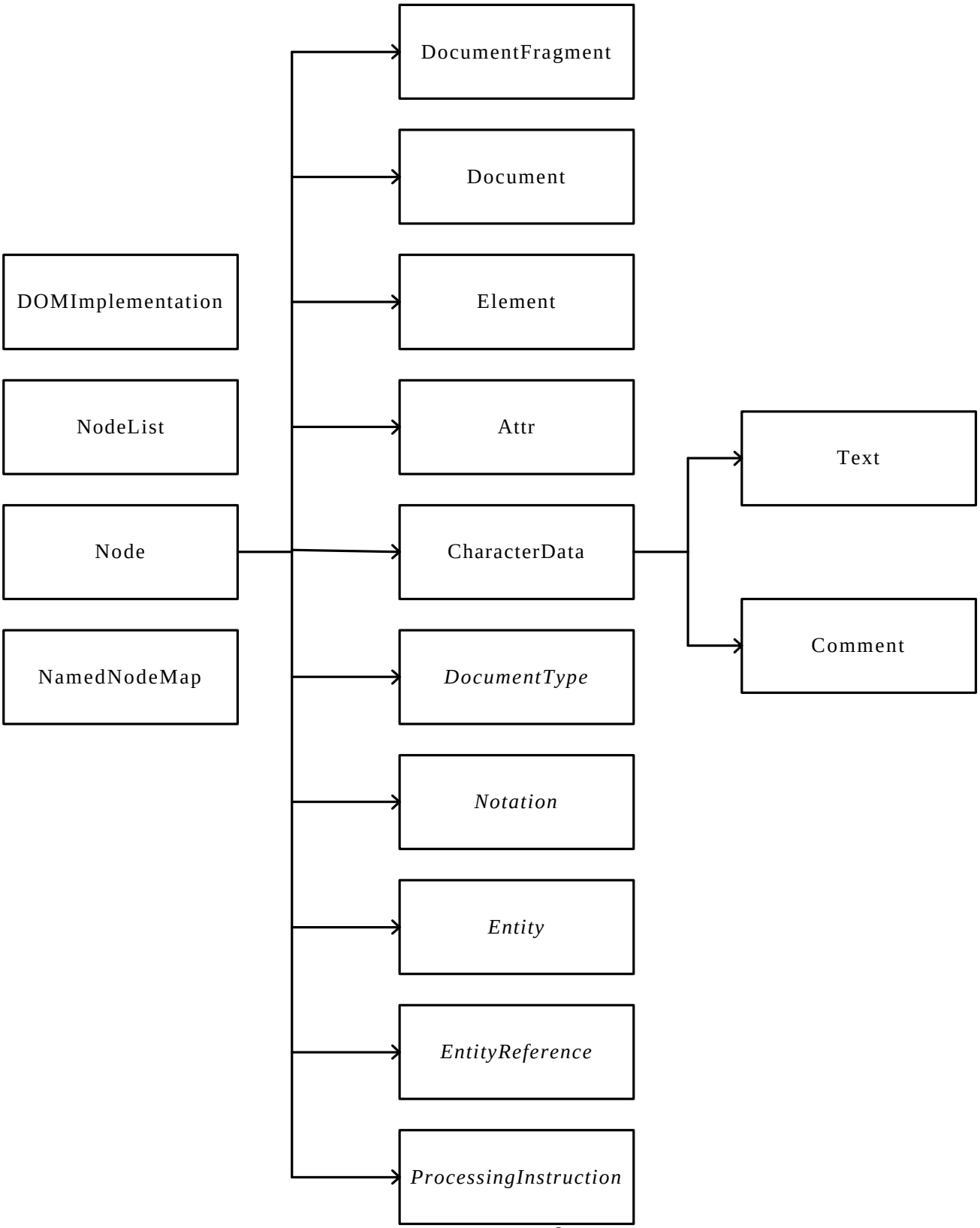


Figure No. 11 DOM Core [2]

In order to manipulate any XML document, we need to know how these interfaces work. In the following section, only the most fundamental interface in DOM has been explored. We shall look at a couple of commonly used properties of the particular interface.

The most basic interface in the DOM is the Node. Almost all objects extend this interface, since any part of an XML document is a node. Using the DOM technology, the XML document is viewed as a tree structure. The root of the tree for our Sponsor list is the document element <Sponsors>. Almost every node has a parent node, and almost every node can have one or more child nodes.

It is important to remember that the node construction is an interface. It is not an object (technically, a class) in and of itself [2]. Instead, the objects that our scripts will interact with will implement this interface. This means that all of these different objects, such as text nodes, element nodes, attribute nodes, and so on, will have common properties, methods and structures, but they are still considered different types of objects. Keeping this in mind, lets delve into some methods of the node interface that are useful to us.

Given below is a simple XML document along with code that we could use to access the elements in that document. The JavaScript code makes use of the properties given for the Node interface.

1. <Sponsor>

<Name>Mr. Jim's Steakhouse</Name>

</Sponsor>

Let us assume for the sake of simplicity, that this comprises our Sponsor.xml document.

2. We create a simple web page with the following HTML and JavaScript

```
<HTML>
<HEAD><TITLE>DOM Demo</TITLE><HEAD>
<SCRIPT language="JavaScript">
        var ObjDom;
        objDom = new ActiveXObject("MSXML.DOMDocument");
        objDOM.load("Sponsor.XML");

        // OUR CODE WOULD GO HERE
</SCRIPT>
<BODY> Demo page for DOM Capabilities</BODY>
</HTML>
```

This page is saved with the name Demo.html in the same directory where Sponsor.xml resides. When I created the initial HTML file, I had to load the XML document into Microsoft's DOM implementation, MSXML. I did this using the extensions provided to the DOM. The load( ) method takes a URL to a file and loads it.

3. Using the nodeName and nodeValue properties of the node interface.

Two pieces of information that we would like to have from any type of node are its name and its value. Node provides the nodeName and nodeValue attributes to retrieve this information.

If we have a variable names ObjNode referencing our name element in the Sponsor.xml document, then the following piece of code:

```
//OUR CODE WOULD GO HERE
var ObjNode;
ObjNode = ObjDom.documentElement.firstChild;
```

alert(ObjNode.nodeName);

This would return the name of the node, in our case the “Name” element.

Similarly,

alert(ObjNode.nodeValue);

would return the value of the element, “null”. The reason it returns “null” instead of “Mr. Jim’s Steakhouse” is that the text inside the element is not part of the element itself. It actually belongs to a text node, which is a child of the element node.

I have not used all the interfaces provided in the DOM core. All the interfaces and their properties that were used for this work are documented in Appendix A.

Many programmers use DOM when working with XML documents. However there is a limitation to the capabilities of DOM. Because the DOM is creating all of these objects in memory, one for each and every node in the XML document, implementations can get quite large and take up huge chunks of memory [4]. In the following chapter, we shall explore another approach that can be deployed if the DOM implementation is too slow.

6. The SAX Architecture

When it comes to analyzing XML documents, DOM is not the only available solution. There is another API called Simple API for XML (SAX), which turns out to be more efficient at certain things in which DOM lacks performance [2]. A typical problem is that the XML documents get too large to load the entire document tree into memory, which takes its toll on performance. The solution to such a problem would be SAX. SAX is not a W3C recommendation, but was created by members of the xml-dev mailing list led by David Megginson.

6.1 The SAX Approach

The Simple API for XML (SAX) is an industry-standard API intended for high-performance XML document processing. It was developed to enable more efficient analysis of large XML documents. To get a better understanding of the SAX way of working with XML, lets compare the two approaches.

A DOM parser processes a node, it places it in the next position of a hierarchical tree, and so the entire data structure is available while processing any node. You can perform sophisticated processing, but you must keep the entire structure in memory, an expensive proposition when the XML structure has tens of thousands of nodes.

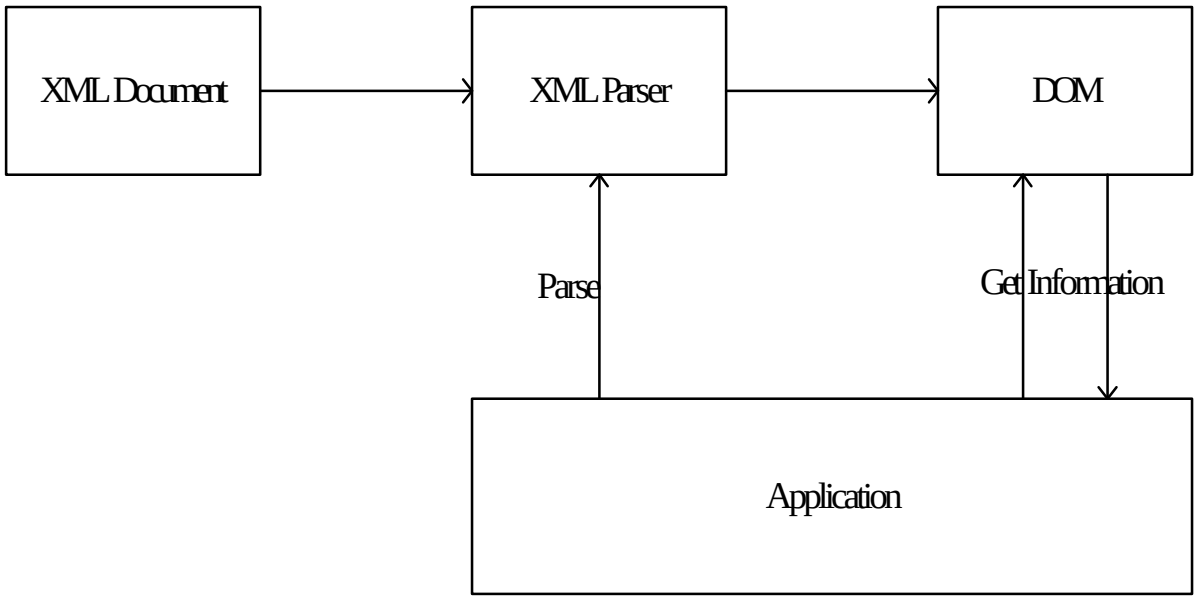


Figure No. 12 The DOM approach. [2]

A SAX parser is remarkably efficient, taking very little memory and not much processor time to stream content. The SAX definition of state is basically declarative the parser only knows about the current node, and can process only those nodes that are either younger siblings or children of the current node [4].

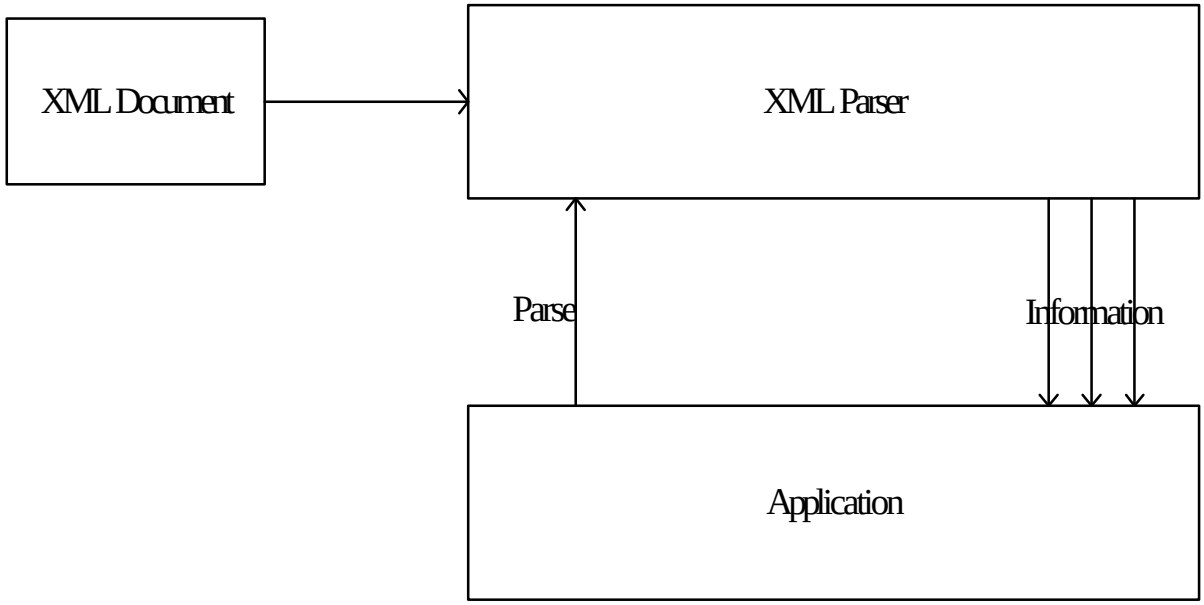


Figure No. 13 The SAX approach. [2]

SAX (Simple API for XML) is an event-driven model for processing XML. Most XML processing models (for example: DOM) build an internal, tree-shaped representation of the XML document. The developer then uses that model's API (getElementsByTagName in the case of the DOM) to access the contents of the document tree. To achieve its goals, SAX uses event-based processing. A SAX parser reads from the input XML document and notifies the application of interesting events. For example, the parser may read an element tag from the input document, and then notify the application through the startElement event. After the event completes, the parser is not required to retain this element tag and associated information in memory.

In order to get SAX, there are two things that are needed: a SAX aware parser and the SAX Java classes. These are the classes that shall build on our application in order to receive SAX events. Java was chosen as the programming interface, due to robust features and platform independence and portability of code. At the time of writing, no web browsers are available that contain a SAX aware parser.

I have provided detailed step-by-step instructions for installing SAX and XP, the SAX aware parser in Appendix B.

6.2 Working with SAX

The SAX model is quite different since it is an event based API (Application Programming Interface). An event-based API reports parsing events (such as the start and end of elements) to the application using callbacks [14]. The application implements and registers event handlers for the different events [14]. Code in the event handlers is designed

to achieve the objective. The process is similar (but not identical) to creating and registering event listeners in the Java Delegation Event Model.

There are three classes of event handlers: DTDHandlers, for accessing the contents of XML Document-Type Definitions; ErrorHandlers, for low-level access to parsing errors; and, by far the most often used, DocumentHandlers, for accessing the contents of the document. In the Java application that has been developed, only the DocumentHandler and ErrorHandler interfaces were used. I shall cover some methods of the DocumentHandler interface that have been used in developing the application. The detailed Java code, along with the front end user interface is provided in Appendix E.

A SAX processor will pass the following events to a DocumentHandler:

- The start of the document.
- A processing instruction element.
- The beginning of an element, including that element's attributes.
- The text contained within an element.
- The end of an element.
- The end of the document.

As seen above the DocumentHandler interface contains a series of methods. SAX provides us with a standard implementation of all the above methods, in class called HandlerBase. So instead of extending the DocumentHandler interface, we can extend the HandlerBase interface, like this:

Public class Parse extends HandlerBase;

In order for us to use the HandlerBase class, we need to import org.sml.sax package. This will ensure that the Java compiler knows where to find the classes it cannot find locally.

Once we have the code developed for instantiating objects of Parse class, all that is really left to do is to write methods that would receive events coming back from the parser.

Some methods for catching events are explained below:

Public void startDocument( ) throws SAXException

This method allows us to receive notification at the start of an XML document. Similarly there is an endDocument method that provides notification when the parser is done parsing the entire XML document.

To receive notification at the start and end of each element, we have the following methods respectively:

Public void startElement(String name, AttributeList attributes) throws SAXException

Public void startElement(String name) throws SAXException

In addition to the above methods, many other methods have been used which are documented in the code in Appendix E. Error handling has been used to make the program more user friendly. The front – end user interface was developed using the Abstract Window Toolkit (AWT) provided in the Java Development Kit [14].

6.3 Limitations of using SAX

SAX is a great tool for analyzing and extracting content from XML documents. In the introduction to this chapter, we addresses some key issues where SAX tends to beat its counterpart DOM. However there are certain situations in which the use of SAX undermines the strength of the application. Here are a few:

- There is no control over the order in which the parser searches. This implies that you may need to build up data that you need over several event notifications [2].
- SAX gives complete information about the active content of the document, but lacks in providing other details such as order of attributes in the element, and comments [4].
- Its read only meaning you cannot adjust the elements of the document. The only way to create a new document is to catch all events and then regenerate the document.
- Another minor disadvantage is that none of the current web browsers support SAX implementation. Currently SAX is limited to server side applications.

In spite of all the limitation listed above, SAX is an excellent API for analyzing and extracting information from large XML documents without incurring the time and overheard associated with DOM.

7. Architecture of the final system.

The information collected resulted in a list of sponsors and parts for Bearcat III. This was then structured into an XML document and enforced using a DTD. The XML document was required to follow the rules and hierarchies established by the underlying DTD. A key decision was the choice of the parser – one of XML's core technologies. Due to the complementary nature of DOM and SAX, I decided to implement both the parser API's for developing the final application.

7.1 The DOM implementation

The DOM API was implemented using Microsoft's MSXML package that ships with Internet Explorer 5.0. This application would allow any user having access to the World Wide Web to draw information about the part list as well as the sponsor list. The DOM model would thus serve as an interface on the World Wide Web to access data about Bearcat III.

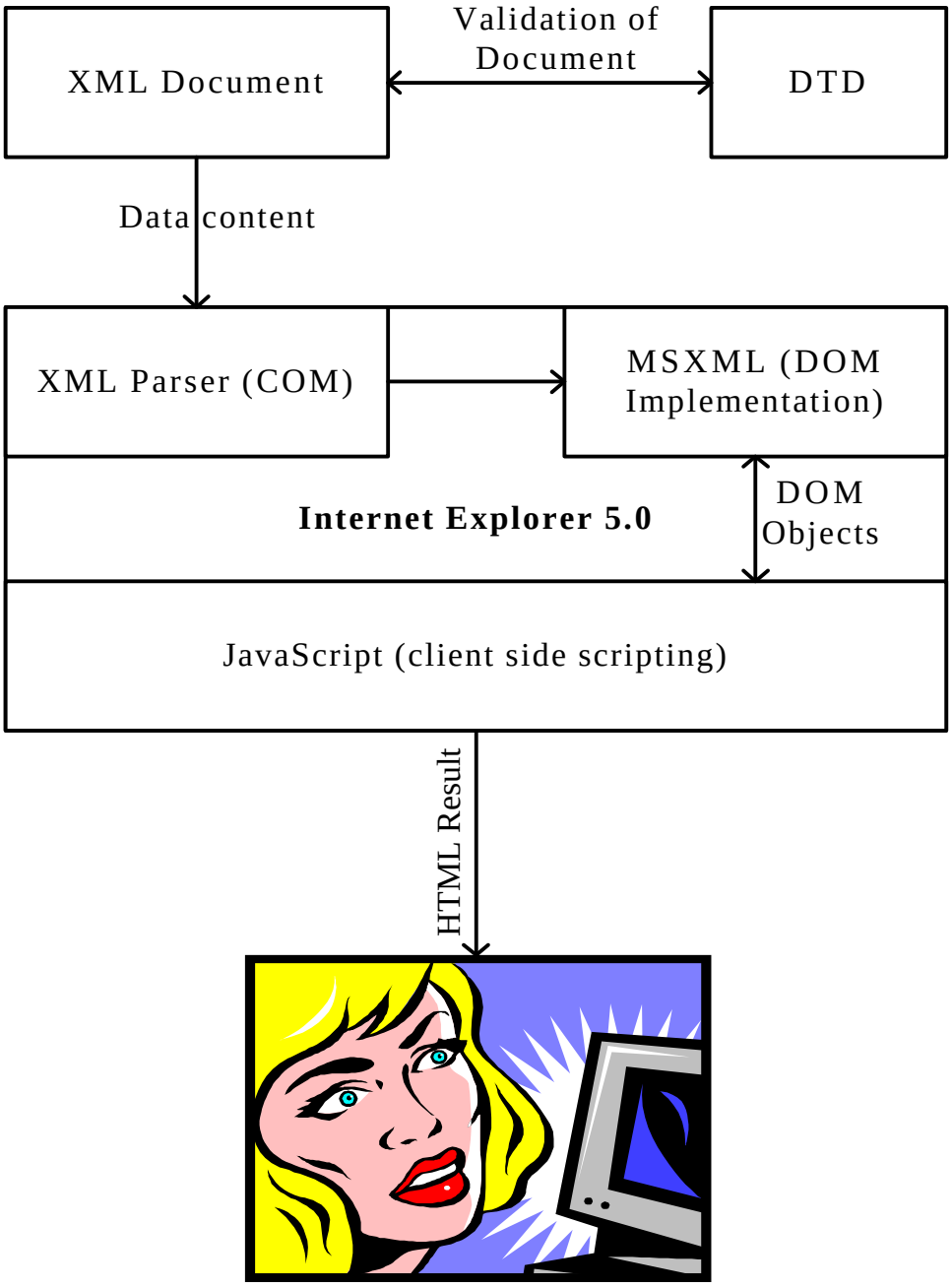
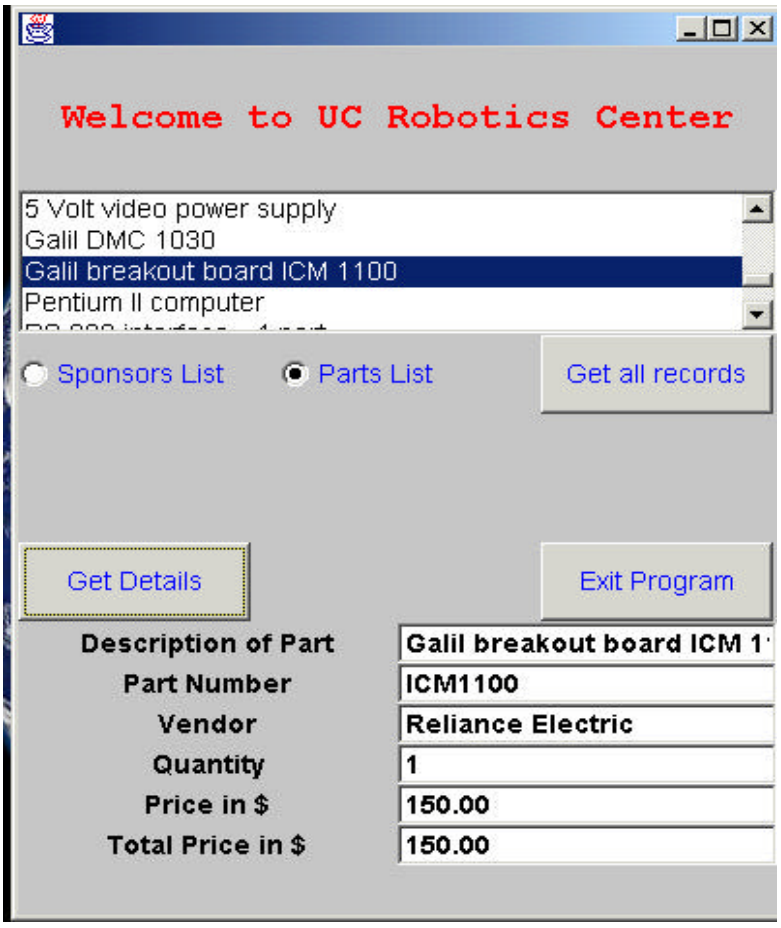


Figure No. 14 Architecture of the DOM Implementation

7.2 The SAX Implementation.

The XP parsers as well the SAX API classes provide a variety of classes to analyze the XML document. A programming language, Java in my case was used to work with objects and produce an output through a user-friendly interface. The front-end interface was developed using Abstract Window Toolkit (AWT) classes. Using the event delegation model approach, objects were registered to get notification in the case of certain events taking place, such as clicking of submit button. This would trigger a series of events, which would result in the parsing of XML document and display of the required information. The front end user interface is shown below:



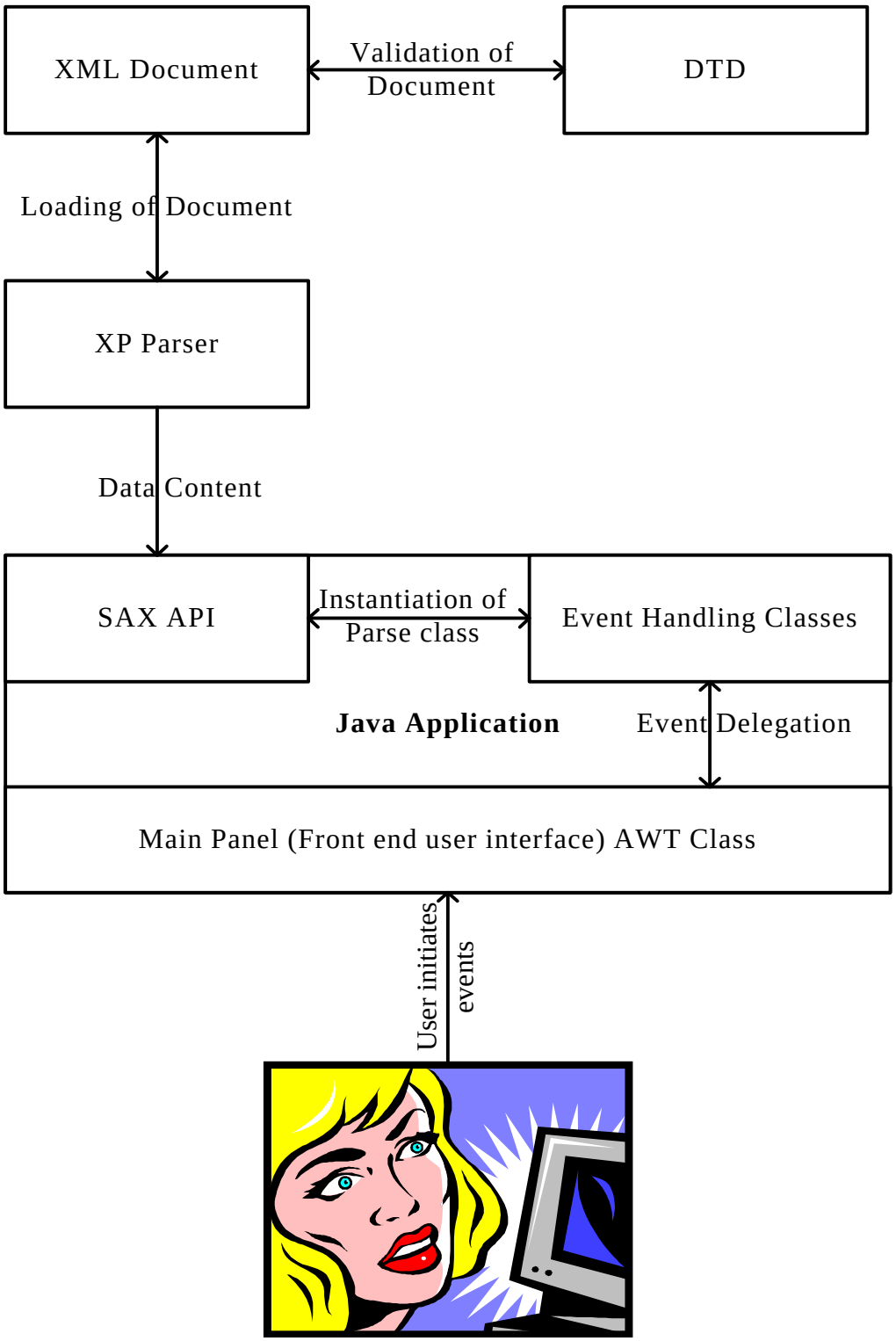


Figure No. 15 The SAX Architecture

The user interface was developed as a combination of three panels, which were laid out using a grid layout approach. Each panel has number of AWT objects that are registered to receive specific events. This modular approach allows for scalability of this application, if needed in the future. The user communicates with the application by clicking on various buttons. These actions result in the generation of an event that is passed to all objects that are registered to listen to that particular event. The user is presented with the choice of being presented with either the Sponsor list or the Robot parts list. Once the user makes a selection, an event gets fired that makes the “Get all records” button visible on the main panel. This button is used to retrieve all records from that specific XML document. This data is displayed in a scroll list. Once the user makes a selection and hits the “Find” button, the event triggers instantiation of a parse class object. This object in turn loads the XML document, parses through it with the help of XP parser and displays the result in the text fields of the main panel. The flow of events is documented in this flowchart below:

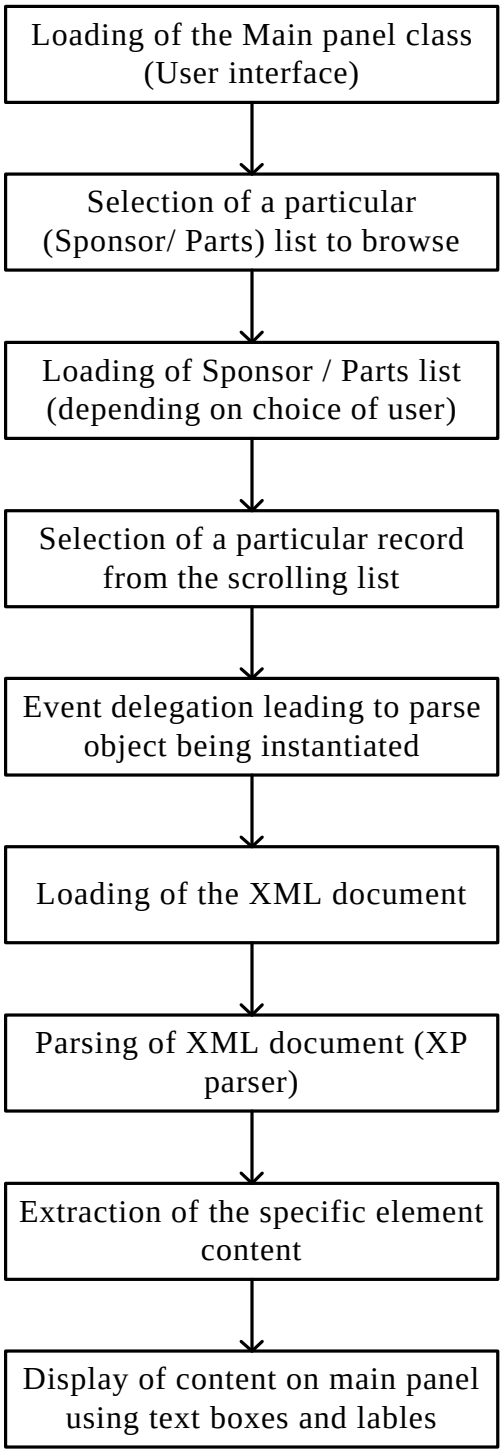


Figure No: 16 Flow of Events

The entire code for this program has been provided in Appendix E.

8. Conclusions and Future Recommendations

The online catalog system has been developed and shall be available on the World Wide Web at the University of Cincinnati Robotics home page. However Internet Explorer 5.0 shall be the only qualified browser to interpret this XML document and display it in a user-friendly format. In the near future, other browsers would come with XML aware parsers built in them. Till then, the only limitation of the web application is the use of a specific browser.

On the other side the Java application that has been developed can be deployed on any platform provided they have the Java Virtual Machine built into them. The Java application can extract data and feed it into other applications and generate useful outputs such as mailing labels. Java is all about portable code, and XML is portable data. The combination of these two technologies would result in true "portable objects" which would form the basis of large-scale distributed systems [12]. Since the web is aiming to become such a system, it needs both these powerful technologies [12].

Some future applications of XML, and areas where the shortcomings of HTML would become evident are:

1. Applications that require the Web client to mediate between two or more heterogeneous databases [13].

2. Applications that attempt to distribute a significant proportion of the processing load from the Web server to the Web client [12].

3. Applications that require the Web client to present different views of the same data to different users [13].

4. Applications in which intelligent Web agents attempt to tailor information discovery to the needs of individual users [1].

As seen above, XML has a great future and is definitely the way to go. Future team members could build upon in this concept of portable data and develop applications that would provide for more efficient and reliable access of information. Just recently XML Schema has become a W3C recommendation. XML Schemas define shared markup vocabularies, the structure of XML documents that use those vocabularies, and provide hooks to associate semantics with them [13]. The use of XML Schemas would provide for better structuring of data and strengthen the performance of applications using the information.

Currently, there are attempts going on to interface Bearcat III with the World Wide Web using wireless technology. This would allow remote invocations of the vehicle, providing the grounds for a truly distributed system. In a sense, future team members would be able to work on the vehicle from their computers at home, or who knows from some wireless device. The same concept of structuring and presenting data in XML can be extended to a Wireless Markup Language (WML). WML is a subset of XML and is intended for use in narrowband devices.

Rahul:

You have done o good job on a limited part of the robot. However, how would you extend this to the entire web site of information we now have on the robotics web

page. This includes theses, PPTs, drawings, etc. Could XML handle this or is it mimited to just data bases of well defined classes?

Are these the only references that exist? Have you done a thorough literature search at the library? Are there XML journals?

References

[1] "Guidelines for using XML for Electronic Data Interchange." Version 0.05 25th January 1998. Contributors: Members of the XML/ EDI working group. XML/ EDI Group Home Page URL: http://www.xmledi.org

[2] David Hunter, Kurt Cagle, Dave Gibbons, Nikola Ozu, Jon Pinnock, Paul Spencer, "Beginning XML". Wrox Press Ltd. August 2000.

[3] "The Evolution of User Interface Markup Languages." Rohit Khare, University of California at Irvine, March 17, 2000.

[4] McLaughlin Brett, "Java and XML". O'Reilly & Associates, Inc. June 2000.pages

[5] Ted Gesing, Jeremy Schneider, "JavaScript for the World Wide Web". Peachpit Press. 1997. pages

[6] "Bearcat II Design Report". 8th Intelligence Ground Vehicle Competition. University of Cincinnati Robot team.

[7] “Bearcat III Design Report”. 9th Intelligence Ground Vehicle Competition. University of Cincinnati Robot team.

[8] The University of Cincinnati Robotics team homepage. http://www.robotics.uc.edu/

[9] Vishnuvardhanaraj Selvaraj, “Failure Mode Analysis of an Autonomous Guided Vehicle using JDBC”, Masters Thesis 2001.

[10] Meyyapa Ganesh Murugappan, “Online Obstacle Avoidance System for an Autonmous Guided Vehicle”, Masters Thesis 2000.

[11] Karthikeyan Kumaraguru, “eDesigning Automated Guided Vehicles”, Masters Thesis 1999.

[12] “XML and Java Technology”, An Interview with Dave Brownell. http://java.sun.com/features/1998/12/Q_A.xml.html

[13] W3C, The World Wide Web Consortium. http://www.w3c.org

[14] Jamie Jaworski, “Java 2 Unleashed”. Techmedia. 1999.pages

Please put in your data bases as well.

Appendix A

XML DOM Interfaces

The following tables list XML DOM programming interfaces. I have included only those properties and methods that have been used in code.

Node

It is the base interface upon which most of the DOM objects are built. It contains methods and attributes that can be used for all types of nodes. This interface includes support for data types, namespaces, DTDs, and XML schemas.

Property	**Description**
nodeName	The name of the node.
nodeValue	The value of the node.
nodeType	The type of the node.
parentNode	The node that is the current node's parent.
childNodes	A NodeList containing all of this node's children.
firstChild	The first child of this node.
lastChild	The last child of this node.
previousSibling	The node immediately preceding the current node.
nextSibling	The node immediately following the current node.

Document

An object implementing this interface represents the entire XML document. This object could be used to create other nodes at run time.

Property	*Description*
Doctype	Returns a DocumentType object, indicating the document type associated with this document.
DocumentElement	The root element for this document
Method	**Description**
CreateElement(tagName)	Creates an element, with the name specified.
CreateTextNode(data)	Creates a Text node, containing the text in the data.
GetElementsByTagName(tagname)	Returns a NodeList of all elements in the document with this tagname.

NodeList

A NodeList contains an ordered group of nodes, via an integral index.

Property	**Description**
Length	The number of nodes contained in this list.
Item(index)	Returns the Node in the list at the indicated index.

NamedNodeMap

This interface represents an unordered collection of nodes, retrieved by name.

Property	*Description*
Length	The number of nodes in the map.
GetNamedItem(name)	Returns a Node, where the nodeName is the same as the name specified.

Item(index)	Returns the Node at the specified index.

Given below is the html file that contains the DOM implementation for the Parts list. The DOM implementation for the Sponsor list is exactly identical. The script in the html file is written using JavaScript.

```
<html>

<head>
<title>DOM</title>
</head>

<SCRIPT Language="JavaScript">

var ObjDom;
ObjDom = new ActiveXObject("MSXML.DOMDocument");
ObjDom.async = false;
ObjDom.load("Parts.xml");

</SCRIPT>

<body>
<font color="#FF0000" face="Georgia" size="3">

 <table border="1" width="100%" height="44"
 bgcolor="#CCFFFF"  bordercolor="#C0C0C0">
<SCRIPT>
var ObjNode;
var ObjNodeList;
ObjNodeList = ObjDom.getElementsByTagName("Part");
var length = ObjNodeList.length;

ObjNode = ObjNodeList.item(0).firstChild;
document.writeln("<tr>");
for(var a=0; a < 6; a++)
{
```

```
  document.writeln("<td>");
  document.writeln("<b>");
  document.writeln(ObjNode.nodeName);
  document.writeln("</b>");
  document.writeln("</td>");
  ObjNode = ObjNode.nextSibling;
}
document.writeln("</tr>");

for(var count=0; count < length; count++)
{
  ObjNode = ObjNodeList.item(count).firstChild;
  document.writeln("<tr>");
  for (var e=0; e < 6; e++)
   {
    ObjNode = ObjNode.firstChild;
    document.writeln("<td>");
    document.writeln(ObjNode.nodeValue);
    document.writeln("</td>");
    ObjNode = ObjNode.parentNode;
    ObjNode = ObjNode.nextSibling;
   }
   document.writeln("<p>");
   document.writeln("</tr>");

}
</SCRIPT>


 </table>
</font>


</body>

</html>
```

Appendix B

A. Install JDK

The Java 2 SDK, Standard Edition, v 1.3 is available for free at http://www.java.sun.com/j2se/ . Once this downloaded to the machine, run the setup program and it installs easily. It comes as a self-extracting executable, and all you have to do is follow instruction on the screen. The default directory for jdk is c:/jdk1.3/

B. Install SAX

We shall get SAX from http://www.megginson.com/SAX/. Download the file that is for Java core. It shall be a zip file.

Create a new directory in a convenient place, called something like XML. Create another directory underneath it called SAX.

Under Windows, SAX 1.0 comes as a zip file. Simply extract all the files in it into the SAX directory.

The top level SAX directory contains a whole load of .java files. Delete them since their presence might confuse the compiler.

Inside the javadoc folder, there would be a set of .html files . This is the official documentation for SAX classes.

C. Install XP

We shall be using the XP, a parser developed by James Clark. This is available for free at http://www.jclark.com/xml/xp/. Download the zip file for the parser.

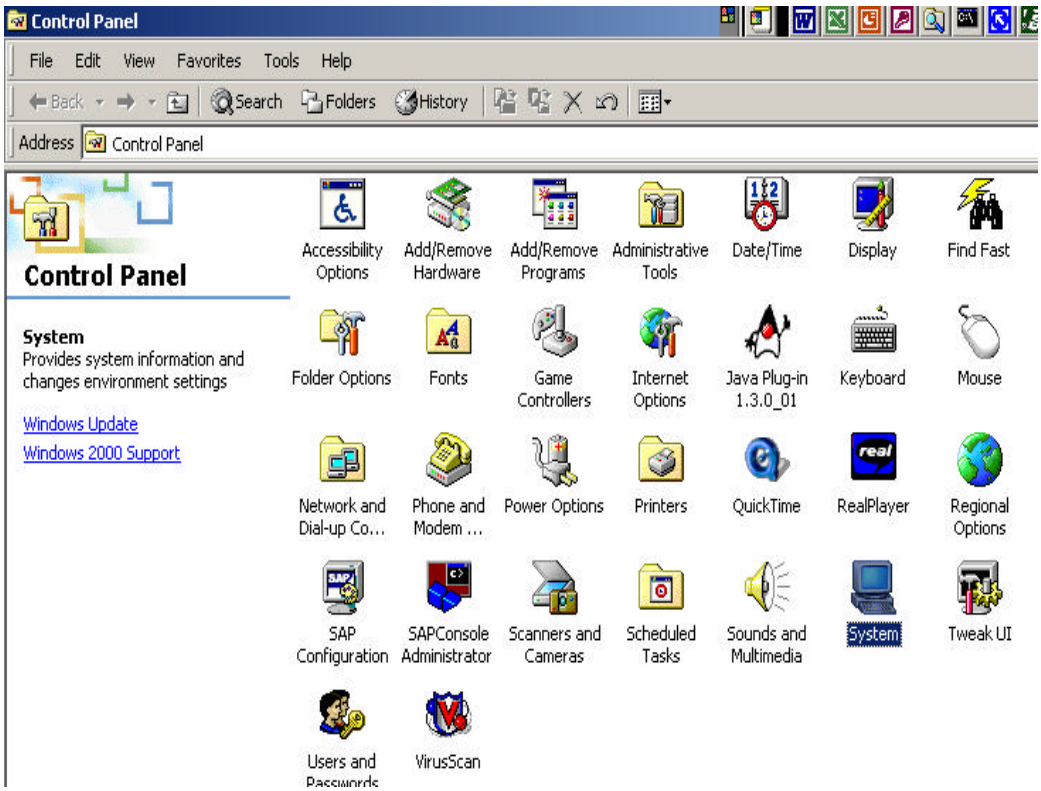
Create another directory under XML, giving it the name xp.

Extract all the files from the downloaded zip file into the xp directory.

D. Modifying the Class Path and Path variables

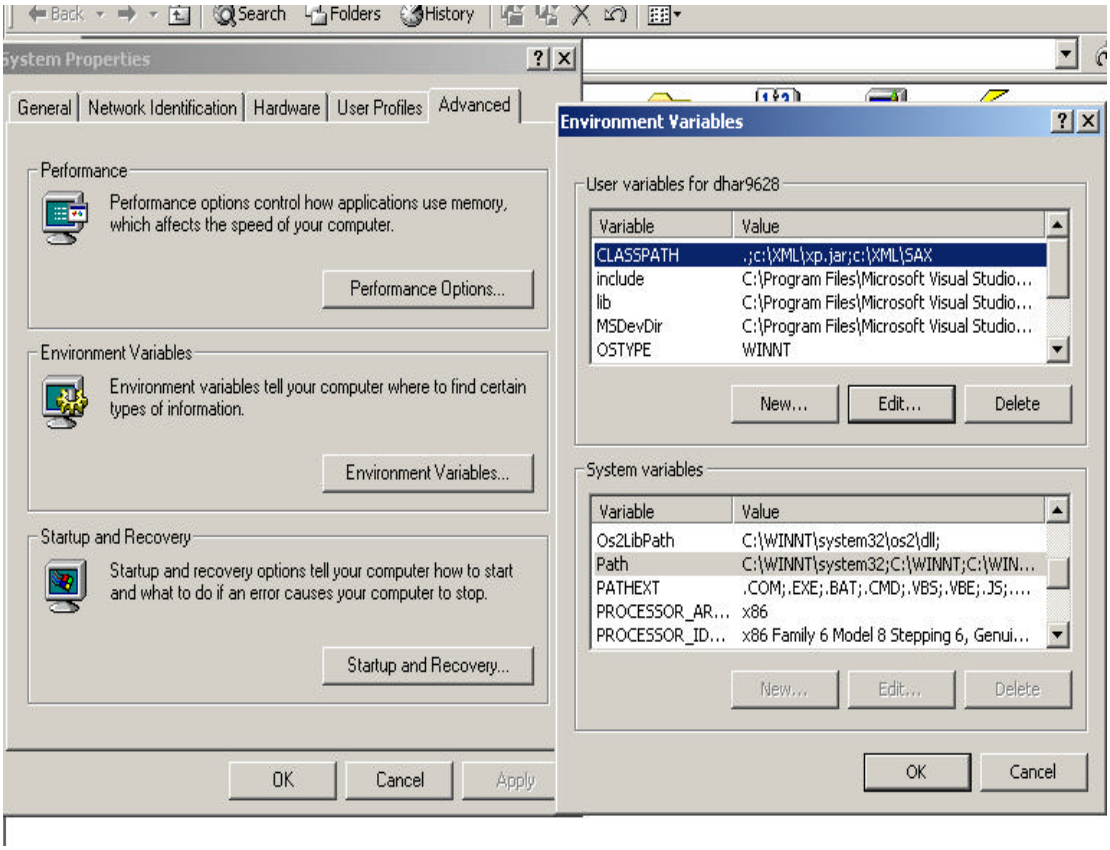
1. Under Windows, this can be done as follows.

   Start → Settings → Control Panel → System → Advanced → Environment Variables.



2. CLASSPATH would be in the User Variables section. You shall need to edit it (or create a new variable) so that it includes the top – level SAX directory and the full pathname of the xp Java archive. The Class Path should also contain the current

directory, ".". This can alternatively be done from DOS prompt using the classpath command.



3. To execute the JDK tools, you need to specify the path to the tool from the command prompt. This requires a change in the PATH statement. The PATH can be found under system variables section. Edit the PATH to include c:\jdk1.3\bin. This can be done from the command prompt using the following command:

   Set path = %path%;c:\jdk1.3\bin

Appendix C

Given below is the Document Type Definition for the Sponsor list

```
<!ELEMENT Sponsorers (Sponsor) >

<!-- <Sponsor> Section -->
<!ELEMENT Sponsor (Name, Company?, Type, Details*, Address?,
City?) >

<!ELEMENT Name (#PCDATA) >
<!ELEMENT Company(#PCDATA) >
<!ELEMENT Type (#PCDATA) >

<!-- <Details> Section -->
<!ELEMENT Details (Amount, Year) >

<!ELEMENT Amount (#PCDATA) >
<!ELEMENT Year (#PCDATA) >


<!ELEMENT Address (#PCDATA) >
<!ELEMENT City (#PCDATA) >

<!-- ===== END of Sponsorers DTD ===== -->
```

Also below is the Document Type Definition for the Parts list

```
<!ELEMENT Parts (Part) >

<!-- <Part> Section -->
<!ELEMENT Part (Description, Number, Vendor, Quantity,
Price, Total) >

<!ELEMENT Description(#PCDATA) >
<!ELEMENT Number (#PCDATA) >
<!ELEMENT Vendor (#PCDATA) >
<!ELEMENT Quantity (#PCDATA) >
<!ELEMENT Price (#PCDATA) >
<!ELEMENT Total (#PCDATA) >

<!-- ===== END of Parts DTD ===== -->
```

Appendix D

Given below is the Sponsor.XML document.

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE Sponsorers SYSTEM
"file:///c:/Rahul/XML/Sponsorers.dtd">

<Sponsorers>

 <Sponsor>
   <Name>Mr. John J. Judge</Name>
   <Company>R.O.V. Technologies Inc.</Company>
   <Type>Corporate</Type>

   <Details>
     <Amount>100</Amount>
     <Year>1999</Year>
   </Details>

   <Address>P.O. Box 10, Franklin Road</Address>
   <City>Vernon, VT 05354</City>
 </Sponsor>

 <Sponsor>
   <Name>Mr. Anthony F. Avellano</Name>
   <Company>Planet Products Corporation</Company>
   <Type>Corporate</Type>
```

```
   <Details>
     <Amount>200</Amount>
     <Year>1999</Year>
   </Details>

   <Address>4200 Malsbary Road</Address>
   <City>Cincinnati, OH 45242</City>
 </Sponsor>

</Sponsorers>
```

Please note that only two elements from the entire document have been provided here.

Also given below is the Parts.XML document

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE Parts SYSTEM "file:///c:/Rahul/XML/Parts.dtd">

<Parts>

  <Part>
     <Description>Electro-craft motor model
E728</Description>
     <Number>0728-39-003</Number>
     <Vendor>Reliance Electric</Vendor>
     <Quantity>2</Quantity>
     <Price>900.00</Price>
     <Total>1800.00</Total>
```

```
    </Part>


    <Part>
      <Description>ISCAN video tracker</Description>
      <Number>RK446R</Number>
      <Vendor>Iscan Inc.</Vendor>
      <Quantity>1</Quantity>
      <Price>9000.00</Price>
      <Total>9000.00</Total>
    </Part>

</Parts>
```

Once again, only two elements from the entire Parts.XML document have been shown.

Appendix E

Given below is the source code for all the Java classes and files that have been used to develop this application.

Mpanel.java

```
/* This class builds up the front-end interface. It is
built from three
different panels to provide for a modular structure */

// importing the Abstract Window Toolkit package
import java.awt.*;

public class Mpanel extends Frame
{
  // These are three panels that make up the main panel
  public Bpanel b;
  public Lpanel l;
  public Dpanel d;

// constructor for the class
 public Mpanel(String name)
 {
   Container c = new Container();

  // constructor for each of the sub panels
   l = new Lpanel();
   b = new Bpanel(this);
   d = new Dpanel();

  // adding the subpanels to the current container
   c.add(l);
   c.add(b);
   c.add(d);

  // setting the layout on the panel using the methods of
Layout Manager
   c.setLayout(new GridLayout(3,1));
```

```
   // setting the size, font and background color for this
panel
   setSize(400,500);
   setFont(new Font("Georgia", Font.PLAIN, 14));
   setBackground(Color.lightGray);
   add(c);

  // making the panel visible to the user
   show();
  } // end of Mpanel constructor

 // The main program that instantiates an object of this
class
  public static void main(String args[])
  {
    Mpanel mp = new Mpanel("Interface");
  }
  // end of main program

} // end of Mpanel class
```

Dpanel.java

```
/* This class creates a panel that contains textboxes and
labels
to display the elements from the XML document */

// importing the Abstract Window Toolkit package
import java.awt.*;

public class Dpanel extends Panel
{
  // defining variables for this class
  public TextField a[] = new TextField[7];
  public Label A, B, C, D, E, F, G;

 // constructor for this class
  public Dpanel()
  {
    // creating an array of text fields by calling the
constructor
    a[0] = new TextField();
    a[1] = new TextField();
    a[2] = new TextField();
    a[3] = new TextField();
```

```
    a[4] = new TextField();
    a[5] = new TextField();
    a[6] = new TextField();

    // creating new display lables by calling the
constructor
    A = new Label("Sponsors Name", Label.CENTER);
    B = new Label("Company Name", Label.CENTER);
    C = new Label("Type of Sponsorer", Label.CENTER);
    D = new Label("Amount", Label.CENTER);
    E = new Label("Year", Label.CENTER);
    F = new Label("Street Address", Label.CENTER);
    G = new Label("City and Zip", Label.CENTER);

    // setting the layout for this panel by using methods
of Layout Manager
    setLayout(new GridLayout(7, 2));
    setFont(new Font("Georgia", Font.BOLD, 14));

    // adding the text fields and labels to the panel
    add(A);
    add(a[0]);
    add(B);
    add(a[1]);
    add(C);
    add(a[2]);
    add(D);
    add(a[3]);
    add(E);
    add(a[4]);
    add(F);
    add(a[5]);
    add(G);
    add(a[6]);

  } // end of constructor for the Dpanel class


} // end of class
```

Bpanel.java

```
/* This panel constructs a panel containing buttons
```

```
These buttons are used by the user to interact with the
back end program */

// importing the Abstract Window Toolkit package
import java.awt.*;

public class Bpanel extends Panel
{


  public Button Find, Clear, Exit, Get;
  public Checkbox Sponsor, Parts;
  public CheckboxGroup choice;
  public Mpanel mp;

  // constructor for this class
  public Bpanel(Mpanel m)
  {
    mp = m;

    choice = new CheckboxGroup();

    Sponsor = new Checkbox("Sponsors List", true, choice);
    Parts = new Checkbox("Parts List", false, choice);

    Get = new Button("Get all records");
    Find = new Button("Get Details");
    Clear = new Button("Clear Fields");
    Exit = new Button("Exit Program");

    // Setting this button to invisible
    Find.setVisible(false);
    Clear.setVisible(false);
    Exit.setVisible(false);

    // registering these objects for receiving events from
the user
    Exit.addActionListener(new Exit(mp));
    Find.addActionListener(new Find(mp));
    Get.addActionListener(new Slist(mp));

    // creating a layout for this panel
    setLayout(new GridLayout(2, 3, 15, 70));

    setForeground(Color.blue);

    // adding the elements to this panel
```

```
    add(Sponsor);
    add(Parts);
    add(Get);
    add(Find);
    add(Clear);
    add(Exit);
  }// end of constructor for this class

} // end of class
```

Lpanel.java

```
/* This panel creates a panel containing a label and a
scrolling list
This panel is combined into the main panel class */

// importing the Abstract Window Toolkit package
import java.awt.*;

public class Lpanel extends Panel
{

  public Label welcome;
  public List display;
  public String choice;

  // constructor for this class
  public Lpanel()
  {
    display = new List(3);

    // making this scroll list invisible
    display.setVisible(false);

    welcome = new Label("Welcome to UC Robotics Center",
Label.CENTER);
    welcome.setFont(new Font("Courier", Font.BOLD, 20));
    welcome.setForeground(Color.red);

    // creating a layout for this panel
    setLayout(new GridLayout(2,1));

    // adding the elements to this panel
    add(welcome);
    add(display);
```

```
  } // end of constructor for this class

  /* This method returns a string that contains the user's
choice
     from the scroll list */
  public String getItem()
  {
    choice = display.getSelectedItem();
    return choice;
  } // end of getitem method


} // end of class
```

Find.java

```
/* This class responds to events dispatched by the Find
button
Depending on the choice of the list, and the item selected.
This class retrieves details from the XML document
*/

// Importing the packages for event handling and parsing
import org.xml.sax.*;
import java.awt.*;
import java.awt.event.*;

public class Find implements ActionListener
{
  private Mpanel me;
  private String find;
  private Parse s;
  private Pparse p;

  // constructor for this class
  public Find(Mpanel m)
  {
    me = m;
  } // end of the constructor

  // method that performs event delegation
  public void actionPerformed(ActionEvent ae)
  {
    // retrieving the choice of the user
    find = me.l.getItem();
```

```
     // using try catch block for error handling
     try {
      /* depending on the choice of the user, the
         corresponding parse class is called */
      if(me.b.Sponsor.getState())
        {
          s = new Parse(find, me);
        }
      else
          p = new Pparse(find, me);
        } catch (Exception ex)
        {
            System.out.println(ex);
        }

  } // end of actionperformed method

} // end of this class
```

Exit.java

```
/* This class terminates the program.
It is registered to listen to events delegated
by the exit button */

// importing the AWT event handling package
import java.awt.event.*;


public class Exit implements ActionListener {

    private Mpanel m;

    // constructor for this class
    public Exit(Mpanel mp)
        {
        m = mp;
     } // end of constructor

    // method that is called on event delegation
    public void actionPerformed(ActionEvent ae)
        {
       System.exit(0);
     } // end of method
```

```
} // end of class
```

Slist.java

```
/* this class is registered to receive events from
the "get all records" button. It manipulates the
items of the Dpanel and Bpanel class
*/

// importing the SAX and AWT event handling packages
import org.xml.sax.*;
import java.awt.*;
import java.awt.event.*;

public class Slist implements ActionListener
{
  private Mpanel me;
  private Getslist s;
  private Getplist p;

  // constructor for this class
  public Slist(Mpanel m)
  {
    me = m;
  } // end of constructor

  // this method handles the event delegation
  public void actionPerformed(ActionEvent ae)
  {
     /* the items are made visible / invisible
        text fields are cleared */
     me.b.Find.setVisible(true);
     me.b.Exit.setVisible(true);
     me.l.display.setVisible(true);
     me.l.display.removeAll();
     me.d.a[0].setText("");
     me.d.a[1].setText("");
     me.d.a[2].setText("");
     me.d.a[3].setText("");
     me.d.a[4].setText("");
     me.d.a[5].setText("");
     me.d.a[6].setText("");

     try {
```

```
      // determines which checkbox is selected by the user
      if(me.b.Sponsor.getState())
        {
          // instantiates an object of Getslist class
          s = new Getslist(me);
          me.d.G.setVisible(true);
          me.d.a[6].setVisible(true);
          me.d.A.setText("Sponsors Name");
       me.d.B.setText("Company Name");
       me.d.C.setText("Type of Sponsorer");
       me.d.D.setText("Amount");
          me.d.E.setText("Year");
       me.d.F.setText("Street Address");
          me.d.G.setText("City and Zip");
        }
      else
        {
          // instantiates an object of Getplist class
       p = new Getplist(me);
          me.d.A.setText("Description of Part");
       me.d.B.setText("Part Number");
       me.d.C.setText("Vendor");
       me.d.D.setText("Quantity");
          me.d.E.setText("Price in $");
       me.d.F.setText("Total Price in $");
       me.d.G.setVisible(false);
          me.d.a[6].setVisible(false);

     } // end of else

        } catch (Exception ex)
            {
              System.out.println(ex);
            }

  } // end of the actionperformed method

} // end of this class
```

Getplist.java

```
/* This class loads the part names from the
Parts.XML document. This is loaded into the
scroll list object */
```

```
// importing the SAX packages
import org.xml.sax.*;


public class Getplist extends HandlerBase
{
  private StringBuffer Sax = new StringBuffer();
  private boolean Found = false;
  private int count = 0;
  public Mpanel m;

  // constructor for this class
  public Getplist(Mpanel mp) throws Exception
   {
    m = mp;
    // instantiating a parser object
    Parser parserobj = new com.jclark.xml.sax.Driver();
    parserobj.setDocumentHandler(this);

    // loading the Parts.XML document to be parsed
    parserobj.parse("file:///c:/Rahul/TCode/Parts.xml");
   }

  // this method is invoked for the start of each element
in the XML document
  public void startElement(String name, AttributeList atts)
    throws SAXException
   {
    if (name.equals("Description"))
      Found = true;
      Sax.setLength(0);
   }

  // this method is invoked for the end of each element in
the XML document
  public void endElement(String name) throws SAXException
   {

    if(name.equals("Number"))
     {
      Found = false;
     }

    // checking if the element is present
    if(Found)
     {
      if(Sax.toString().trim().equals(""))
```

```
       {
       }
      else {
       // displaying the element in the scroll list
       m.l.display.add(Sax.toString().trim());

       Sax.setLength(0);
       count++;
       } // end of else
     } // end of outer if
   } // end of endElement method

  // this method retrieves the content of each element
  public void characters(char[] ch, int start, int len)
throws SAXException
   {
      Sax.append(ch, start, len);
   }

} // end of this class
```

Getslist.java

```
/* This class loads the Sponsor names from the
Sponsors.XML document. This is loaded into the
scroll list object */

// importing the SAX packages
import org.xml.sax.*;


public class Getslist extends HandlerBase
{
  private StringBuffer Sax = new StringBuffer();
  private boolean Found = false;
  private int count = 0;
  public Mpanel m;


  // constructor for this class
  public Getslist(Mpanel mp) throws Exception
   {
    m = mp;
     // instantiating a parser object
    Parser parserobj = new com.jclark.xml.sax.Driver();
```

```
    parserobj.setDocumentHandler(this);

    // loading the Sponsors.XML document to be parsed
    parserobj.parse("file:///c:/Rahul/TCode/Sponsors.xml");
   }


  // this method is invoked for the start of each element
in the XML document
  public void startElement(String name, AttributeList atts)
    throws SAXException
   {
    if (name.equals("Name"))
      Found = true;
      Sax.setLength(0);
   }

  // this method is invoked for the end of each element in
the XML document
  public void endElement(String name) throws SAXException
   {

    if(name.equals("Company"))
     {
      Found = false;
     }

     // checking if the element is present
    if(Found)
     {
      if(Sax.toString().trim().equals(""))
       {
       }
      else {
       // displaying the element in the scroll list
       m.l.display.add(Sax.toString().trim());

       Sax.setLength(0);
       count++;
       } // end of else
     } // end of outer if
   } // end of endElement method


  // this method retrieves the content of each element
  public void characters(char[] ch, int start, int len)
throws SAXException
```

```
   {
      Sax.append(ch, start, len);
   }

} // end of this class
```

Parse.java

```
/* This class searches through the Sponsor.XML document
to find details about the specific sponsor as requested
by the user */

// importing the SAX package
import org.xml.sax.*;


public class Parse extends HandlerBase
{
  private StringBuffer Sax = new StringBuffer();
  private boolean Found = false;
  private String s, result;
  private int count = 0;
  private int index = 0;
  public Mpanel m;

  /* constructor for this class. Takes two parameters,
     the first being the choice of the user, and second
     an object of the main panel class */
  public Parse(String name, Mpanel mp) throws Exception
   {
    s = name;
    m = mp;
    Parser parserobj = new com.jclark.xml.sax.Driver();
    parserobj.setDocumentHandler(this);

    // loads the Sponsor.XML document that needs to be
parsed
    parserobj.parse("file:///c:/Rahul/TCode/Sponsors.xml");

   } // end of constructor

  // method that is invoked at the start of each element
  public void startElement(String name, AttributeList atts)
    throws SAXException
   {
    if (name.equals("Name"))
      Sax.setLength(0);
   }
```

```
  // method that is invoked at the end of each element
  public void endElement(String name) throws SAXException
   {
    // checks for matching element content with user choice
    if(Sax.toString().equals(s))
      Found = true;

    if(name.equals("Sponsor"))
     {
      Found = false;
     }

    if(Found)
     {
      if(Sax.toString().trim().equals(""))
         {
           if (count == 1)
             {
               m.d.a[index].setText("");
               index++;
             }
           System.out.println(Sax.toString().trim());
         }
      else {
             // sets the text field to display the element
content
               m.d.a[index].setText(Sax.toString().trim());
             Sax.setLength(0);
             index++;
      }
     count++;
     }
   } // end of endElement method

  // extracts the content of the specific element
  public void characters(char[] ch, int start, int len)
throws SAXException
   {
      Sax.append(ch, start, len);
   }

} // end of this class
```

Pparse.java

```
/* This class searches through the Parts.XML document
to find details about the specific Part as requested
by the user */

// importing the SAX package
import org.xml.sax.*;


public class Pparse extends HandlerBase
{
  private StringBuffer Sax = new StringBuffer();
  private boolean Found = false;
  private String s, result;
  private int count = 0;
  private int index = 0;
  public Mpanel m;

  /* constructor for this class. Takes two parameters,
     the first being the choice of the user, and second
     an object of the main panel class */
  public Pparse(String name, Mpanel mp) throws Exception
   {
    s = name;
    m = mp;
    Parser parserobj = new com.jclark.xml.sax.Driver();
    parserobj.setDocumentHandler(this);

    // loads the Parts.XML document that needs to be parsed
    parserobj.parse("file:///c:/Rahul/TCode/Parts.xml");

   } // end of constructor


  // method that is invoked at the start of each element
  public void startElement(String name, AttributeList atts)
    throws SAXException
   {
    if (name.equals("Description"))
      Sax.setLength(0);
   }

  // method that is invoked at the end of each element
  public void endElement(String name) throws SAXException
   {
    // checks for matching element content with user choice
```

```
    if(Sax.toString().equals(s))
      Found = true;

    if(name.equals("Part"))
     {
      Found = false;
     }

    if(Found)
     {
      if(Sax.toString().trim().equals(""))
       {
       }
      else {
             // sets the text field to display the element
content
             m.d.a[index].setText(Sax.toString().trim());

             Sax.setLength(0);
             index++;
              count++;
       } // end of else
     } // end of outer if
   } // end of endElement method


  // extracts the content of the specific element
  public void characters(char[] ch, int start, int len)
throws SAXException
   {
      Sax.append(ch, start, len);
   }

} // end of this class
```