The Structured-Element Object Model for XML

MA Chak Kei

A Thesis Submitted in Partial Fulfillment
of the Requirements for the
Degree of Master of Philosophy
in
Computer Science & Engineering

Supervised by:
Prof. Michael R. Lyu

The Structured-Element Object Model for XML

submitted by

MA Chak Kei

for the degree of Master of Philosophy
at the Chinese University of Hong Kong

Abstract

The Extensible Markup Language (XML) is widely used in exchanging and storing information for various Internet-enabled applications. In general, XML represents semi-structured data with no rigid schema, and the basis for processing XML data is to establish a suitable data model for XML data. There has been a lot of work for modeling XML data into relational and object-relational models to facilitate better information management. However, for XML data modeling in application development, an object-oriented data model can reflect the concepts of components in object-oriented programming languages more appropriately, which benefits building of reusable software components.

Consider that many complex data structures like relational data and various indexing trees are frequently used in many programs, storing their contents in XML would bring a lot of convenience and flexibility. However, application programmers are mainly interested in the high-level information it carries but not all the details in indexing or schema modeling. An abstraction for complex data objects would simplify the programs and shorten the development time. Therefore, we will present an XML data modeling in which the physical data representation and the logical data representation are more flexibly coupled.

In this thesis, we present the *Structured-Element Object Model* (SEOM) for XML data. The model combines features from the Document Object Model and the data binding technologies. In the Document Object Model, the logical data closely reflects the physical data representation in XML file, i.e., the DOM closely resembles the XML data model as in W3C XML specification. Data binding technologies, on the other hand, reflects semantics in XML data by representing a logical entity as a data object, but it does not maintain the XML document structure anymore.

SEOM, however, combines the features of both XML processing models and delivers features from both models. It adopts the tree structure of DOM and introduces an additional node, *Structured-Element* (SElement). The SElement resembles the interface of a DOM Element. It has an internal data structure to model an individual logical entity. It also implements additional interface to allow queries being made to its internal data structure. With the SElement type, SEOM Document brings great convenience for developing XML application by representing complex data objects under a hierarchical organization.

In this thesis, we will cover the Structured-Element Object Model in details, including its data modeling, schema, parsing and querying. We will also describe a web-based XML query system, which is built to demonstrate how the SEOM helps in a real application.

可擴展標示語言(XML)上結構元素物件模型

作者：馬澤祺

修讀學位：哲學碩士

香港中文大學計算機科學及工程學部

論文摘要

在互聯網程式應用上，可擴展標示語言(XML)正被廣泛應用於信息交換與貯存方面。XML 定義了一種無固定資料格式 (Schema)的半結構形態資料，而爲 XML 資料設計資料模式(Data Model)就成了處理 XML 文件所載資料的第一步。其中，以關聯式(Relational)模形或是物件關聯式(Object-Relational)模形這兩種方法去描述及處理 XML 資料已在資訊管理方面已取得不少發展。而在程式設計方面，以物件導向(Object-Oriented)模形去描述資料則當可更貼合物件導向程式中的使用元件和模組的習慣，這將更乎合”可重覆使用軟體元件”的概念。

XML 的半結構形態使它可以方便且有彈性地裝載諸如關聯式資料結構或者不同的索引樹結構等很多不同的複雜資料結構，然而，對於應用程式設計師而言，更重要的是高層次的應用資料，而非細節如索引資料或資料格式等。另一方面，將複雜資料結構抽像化更可簡化程式的設計及縮短編寫程式所需的時間。爲此，我們提出了一種 XML 資料模型，其設計可以讓實體資料結構和邏輯資料結構以更靈活的方式結合。

在這篇論文中，我們提出了應用於 XML 上之結構元素物件模型(SEOM)。這個模

型結合了文件物件模型(Document Object Model)及資料繫結技術(Data Binding)的特點。文件物件模型表現了近似 W3C 定義的 XML 資料模型，其邏輯資料結構相當近似予 XML 檔案的實體資料結構。另一方面，資料繫結技術以資料本身的邏輯結構爲中心，將 XML 資料以資料物件的方式呈現，然而這種方法卻不能保留 XML 中的樹狀結構。

我們提出的 SEOM 是以文件物件模型的樹狀資料結構爲基礎，並引入了新的資料結點 – 結構元素(Structured-Element)。結構元素以文件物件模型中的元素爲本使其能夠與文件物件模型的樹狀資料結構結合，而其內裏則另藏資料結構以模擬程式中的邏輯資料結構。同時，結構元素加入了新的程式介面用以查詢內藏的資料結構。結構元素的引入使 SEOM 文件能夠輕易地在同一樹狀結構下裝載不同的複雜資料結構，從而使 XML 應用程式的開發更爲方便快捷。

在這篇論文中，我們會詳細剖釋結構元素物件模型，包括其資料模型、格式定義、文件剖析程序和資料查詢。另外，我們還會介紹一個網上 XML 查詢系統以展示 SEOM 能怎樣幫助 XML 應用程的的開發。

Acknowledgements

I would like to take this opportunity to express my gratitude to my supervisor, Prof. Michael R. Lyu for his generous guidance and patience given to me in the past two years. His support and encouragement, as well as the inspiring advice are extremely essential and valuable in my research papers (published in IC'2001) and my thesis.

I am also grateful for the time and valuable suggestions that Prof. Irwin King and Dr. Yiu Sang Moon have given in marking my term papers. Without their effort, I will not be able to strengthen and improve my research projects and papers.

I would like to thank Edward Yau and Sam Sze for their insightful advice and enlightening research ideas. Special thanks should be give to my fellow colleagues, Nicky Ng, K.K. Ng, T.Y. Wong, P.M. Ho, Kenny Kwok, Vincent Cheung, Anson Lee, Joe Lau, Anny Chiu, K.F. Jang, who have helped me in solving programming and computer problems, given me encouragement and supports, and given me a joyful and unforgettable university life.

Finally, my thanks must go to C.Y. Lee and H.Y. Lam for their unfailingly support in those times when I wondered if I could make my degree to the end.

Table of Contents

List of Tables

List of Figures

Chapter 1.

Introduction

1.1 Addressing and Manipulating XML Data

The Extensible Markup Language (XML) [1, 6, 38, 43], accounting for its interoperability and open-standard, is widely used in exchanging and storing information for various Internet-enabled applications. The "semi-structure" property of XML allows users to represent different data flexibly and enhances the development of information systems in data interchange and data storage. It provides a common format for expressing both data structure and contents with in plain text. Moreover, its power is strengthened by various kinds of emerging XML-technologies such as XSLT, XPath [39, 42], XQuery, etc. The existence of simple-to-use application programming interfaces (API) such as SAX and DOM also facilitates rapid development of XML-enabled applications.

One characteristic shared by XML applications is that they are working with data heavily [4, 40]. XML allows diverse and rapid-changing data structures to be represented, which eases the life of storing, validating and manipulating the data. For manipulation of XML data in programs, the Document Object Model (DOM) plays a significant role.

The Document Object Model (DOM) [3, 41] is at first designed as a platform- and language- neutral application programming interface (API) for valid HTML and well-formed XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. With the Document Object Model, programmers can build documents, navigate the document structure, and add, modify, or delete elements and contents. That will be convenient to bind an XML document to a DOM structure to access, change, delete or add data to the document with DOM API.

The DOM API models an XML document data in a tree-like data structure, where the XML elements and character data are modeled as data nodes in the tree. Thus the programmers can access and manipulate the XML data easily through the methods provided by the "node" objects. However, under the Document Object Model, the accessing methods for document nodes are defined by the parent/child relationship and the sibling relationship, which would be insufficient and inefficient in many cases. As a result, the XPath is often incorporated into the programming interface to allow flexible accessing of various parts of the XML document.

DOM is good for manipulating XML data and provides a large degree of freedom to the programmers. Based on the W3C XML specification, the DOM and XPath adopt a basic XML data model and offer a direct mapping for XML data types. However, the development effort will be increased dramatically when the data becomes more complex. To simplify the development process, abstracting is a commonly used technique. The Java Architecture for XML Binding (JAXB) [34], for example, compiles an XML Schema to generate class for handling XML data, thus the programmers do not need to parse the XML file into a DOM structure followed by reading the data values out from the DOM structure.

Moreover, one of the mostly mentioned advantages for XML, its “semi-structured data” property, is not well addressed in this basic XML data model. The “semi-structured” property of XML allows it to flexibly represent various kinds of data, ranging from structured tables in relational databases, spatial-indexing structures in tree-like form, to unstructured data in web contents [4, 5, 7]. The basic XML data model, despite its flexibility in capturing all possible structures existing in an XML document, it does overly simplify the document into branches-and-nodes only structure, and does not address the different properties for various data structures. The understanding of document structure is thus left as a job for the application programmers, which is an unnecessary waste of programming efforts.

1.2 The Structured-Element Object Model (SEOM)

XML is a new standard for representing and exchanging data on the Internet. Modeling XML data for Web applications and data management is a hot topic in XML research area.

In general, XML represents semi-structured data with no rigid schema [2]. When XML is used in application development, data access inside an XML document is a major consideration. For huge documents, primitive filtering features (e.g. string matching in XPath) may not be efficient enough, and an indexing structure may be employed to increase the search speed. For an effective indexing, metadata is needed to provide understanding on the contents in the document [7, 8, 10].

Based on the Document Object Model, we proposed the *Structured-Element Object Model* (SEOM) for representing XML data. The model adopts the DOM tree structure with an additional node, the *Structured-Element* (SElement). The SElement represents a logical entity and it embeds an internal data structure for storing extra information, such as the indexing information. The internal data structure itself can also be used to store the data in a more efficient manner than in a generic tree structure. The SEOM Document brings great flexibility in representing complex data objects under a hierarchical organization.

In order to bind the physical XML data representation to the Java classes that implement SElement type, we define a schema [9] for declaring SElement objects. The schema declares an SElement object by specifying how the XML elements are mapped into the target object's internal data.

To demonstrate our work, we implement a web-based XML query system. The query system is based on the Structured-Element Object Model. It adopts a client-server architecture. The server will parse an XML file into SEOM Document. The client will connect to the server, retrieve information on the document and the current node, prompt user to select a query operation, and send the query to the server. Results will be displayed and the user may further navigate the document by selecting a node from the result.

1.3 Relate Research

XML attracts a lot of attention in web application development and data management. To model XML data using an object-oriented approach is a hot topic in XML research area.

Many contributions have been made to the object-oriented model for XML data. For examples, the Object Exchange Model (OEM) [26] in Lore project [13, 21] and the Document Object Model extend from generic semi-structured data to XML data. In OEM, the XML is described as a labeled, directed graph. The nodes in the graph represent the data elements and the edges represent the element-subelement relationship. However, this data model does not implement classes or types definitions, and it does not support encapsulation and object behavior [22] either.

The W3C DOM interface is widely adopted by different programming languages and scripts for manipulating XML data [4]. It parses XML contents into a hierarchical tree of node objects. It defines the interface for manipulating these objects and for some document-level processing. However, the type of defined operations address only the generic aspect of primitive XML node type; it is possible to enhance the interface by introducing a mechanism for defining application-specific object structures and behaviors. The DOM is the basis for us to develop the SEOM, and details of enhancements will be presented from Chapter 3 to Chapter 5.

The Object-Oriented Representation Model (ORM) [15] is another object modeling for XML data. It defines a set of rules and steps to transform DTDs and XML document into the model. It capsulizes elements of XML data and manipulation methods. However, the limitation of DTD makes the model inflexible in adopting changes in data object types.

Another XML data modeling is presented in [20]. It defines a set of translation rules proposed for translating XML data, with or without DTDs, into classes or objects for a mediator. Terminal elements, as well as attributes of elements, in XML data are modeled as attributes of the object. However, it requires users to have the knowledge of the object structure and there is no distinguishability being made between public and private data members.

In our research, we had learned the problems of previously developed models; and we address these problems in our proposed Structured-Element Object Model.

1.4 Contribution

Briefly speaking, we make the following contributions in our research work:

- We have proposed the *Structured-Element Object Model* to encapsulate an XML Element structure as a single extended-Element, named *Structured-Element*, with mappings defined in the schema. We implement the *R-Tree SElement* as an example in the query system implementation, but this mechanism is generic and can be applied to different data structures.
- We have proposed an *SEOM Schema* for mapping XML Element data into SEOM class objects. The schema is generic to different Element structure types and allows more flexible representation of the Element structure in XML data.
- We have proposed a query wrapper technique for the Structured-Element object, which allows programs to retrieve the available query methods from the SElement, to modify the parameter values in the replied message, and to submit the query to the object. The query wrapper technique does not require the user to know the available query methods in advance. It is especially suitable for implementing queries in an interactive XML system.

- We have enhanced the capability of querying by XPath by adding a *Structured-Element query function* to the XPath expression. The enhanced XPath integrates the core features available in Structured-Element. It provides a powerful mechanism to locate and access XML data under the Structured-Element Object Model.
- We have implemented a *Web-based SEOM Document Query System* to demonstrate our work. This query system applies our mechanism and it can make queries to XML data using the Element structure with the structure-specific query methods.

1.5 Thesis Overview

We would explain the contributions described above in details in the coming chapters. First, we conduct an overview of some related work and technologies in Chapter 2. We describe XML, XML Schema, DOM, and XPath there, as they are closely related to and being used intensively in our project.

In Chapter 3, we introduce our Structured-Element Object Modeling, and give and overview on its functional aspect. Chapter 4 will cover the data modeling and schema modeling for the SEOM. In Chapter 5, we will describe the processes in binding XML data to SEOM Document object, as well as in querying data in SEOM Document.

We demonstrate our mechanism with a web-based query system in Chapter 6, and we describe in detail the components of that system and how we apply our mechanism to enhance it for more flexibility.

In Chapter 7, we evaluate the advantages and disadvantages of our approach. We will also address further enhancement on supporting “semi-structure” in our mechanism. We then conclude our work in Chapter 8.

Chapter 2.

Background Technologies

In this chapter, we will present an overview of some XML technologies as they are closely related to our research project. The research project is focused on improving XML usability by enhancing the DOM data structure. We extend the XML Schema to specify the Element structure, which will be used in binding XML data to Java classes. Beside a query wrapper technique we introduce into the system, we also add a query function to the XPath technology and make it a more powerful XML accessing technology.

2.1 Overview of XML

Extensible Markup Language (XML) [43] is a standard developed by the World Wide Web Consortium (W3C). It is a markup language derived from the Standard Generalized Markup Language (SGML). Some design goals for XML are:

- XML shall be straightforwardly usable over the Internet.
- XML shall support a wide variety of applications.
- It shall be easy to write programs which process XML documents.
- XML documents should be human-legible and reasonably clear.

XML is about the description of data. Its specification describes the XML data format and grammar. The specification makes XML to be a data exchange and representation standard.

Our research mainly focuses on using XML to support complex data objects [25] such as “Lists of Lists of Lists” or “Trees of Trees of Trees” or even a heterogeneous type like “Trees of Lists of Trees.” XML provides flexible structure description and is ideal for storing complex data objects. It allows complex data objects to be sent over the Internet and to be shared among different applications. Our research is based on XML and we will address the basic syntax of XML and the XML namespaces in the following sections.

2.1.1. XML Basic Syntax

XML is a textual representation of data. An XML document uses markups to describe its logical structure. A basic component in XML is the element. It is represented as XML tags enclosed in angle brackets. In an XML-aware application, the XML tags can be handled specially depending on the nature of the application. Users can define new tags for their needs. For an well-formed XML document, all XML elements must have a start tag and a close tag, for example, a pair of tags `<name>` and `</name>` may be used to define an XML element that represents someone’s name. Inside an element we may have text, other elements, or a mixture of both.

XML also allows us to add attributes to the elements. XML attributes are defined as name-value pairs in the element’s start tag, whereas the value is a quoted string. For example, in the element `name` we just defined, we may add an attribute `id` to provide an identity to the person associated with the name, i.e., `<name id=″1″>` tells us that this name has the attribute `id` with value equals to “1”.

XML elements may be nested to form a hierarchical structure. As a well-formed XML document has exactly one root element, the document can always be modeled as a tree structure.

2.1.2. Namespaces in XML

When different sets of XML vocabulary definition are shared among a community, it is possible that name collisions happen between some elements in different sets, which should carry different semantics by their design. XML namespaces adopt a compound name syntax to distinguish identical names that belong to different vocabularies.

An XML namespace is a collection of names, identified by a Uniform Resource Identifiers (URI) reference, which are used in XML documents as element types and attribute names. The URI references are considered identical when they are exactly the same character-for-character.

Names from XML namespaces may appear as qualified names, which contain a single colon, separating the name into a namespace prefix and a local part. The prefix, which is mapped to a URI reference, selects a namespace. The combination of the universally managed URI namespace and the document's own namespace produces identifiers that are universally unique.

To use the XML namespace, a namespace attribute has been added to the start tag of an element to give the element prefix a qualified name associated with a namespace. When a namespace is defined in the start tag of an element, all child elements with the same prefix are associated with the same namespace. The namespace attribute is placed in the start tag of an element and has the following syntax:

```
xmlns:namespace-prefix="namespace"
```

In our research project, the proposed the SEOM schema has the following namespace definition:

```
xmlns:seom="http://www.cse.cuhk.edu.hk/~ckma1/SEOM"
```

2.2 Overview of XML Schema

The XML specification not only describes the XML data format and grammar, but also specifies the architecture for handling XML data. XML Processor, also known as the XML parser, is part of the architecture. An XML parser ensures that the presumed XML data is well-formed. XML well-formedness defined in the XML specification certifies that the XML document has a correct structure and syntax.

An XML parser may also be used to check the validity of the user's data structure. Any XML data object is considered a valid XML document if it is well-formed, meets certain further validity constraints, and matches a grammar describing the document's content. The XML parser can use a separate document, generically called a *schema* [44], to describe and validate that instance of XML data.

In our research project, the Structured-Element Object Model is specified as schema. Besides constraining an XML document, with schema we can create an abstract description of the structure of documents. Applications development can be tied to the schema rather than to a proprietary document structure.

Moreover, as the schema enforces the document structure (by a validating engine), document exchange is more convenient since individual applications (the sender and receiver) do not need to check the document validity. This not only promotes sharing of data or model between communities, but also makes it easier for different application to process the same files.

2.2.1. W3C XML Schema

The W3C XML Schema specification [44] is aimed at offering a way to constrain documents using XML syntax. XML Schemas provide a means for defining the structure, content and semantics of XML documents. An XML Schema constrains an XML document by:

- defining elements that can appear in a document,
- defining attributes that can appear in a document,
- defining which elements are child elements,
- defining the order of child elements,
- defining the number of child elements,

- defining whether an element is empty or can include text,
- defining data types for elements and attributes, and
- defining default and fixed values for elements and attributes.

The W3C XML Schema has the namespace URI `http://www.w3.org/2001/XMLSchema` and it is often associated with the prefix `xsd`. Schema Components use the blocks that make up the abstract data model of the schema. Four primary components include simple type and complex type definitions, and element and attribute declarations.

simple type

Simple types refer to atoms of data that cannot be divided in terms of XML schema and we use `data-types` to define the data-type of information in a simple type. There are many built-in `data-types` in the XML Schema specification, some of which are listed in Table 2.1.

Type	**Description**	**Example**
`string`	Represents any legal character strings	`Jacky Ma`
`boolean`	Represents binary logic	`true, false, 1, 0`
`number`	Represents arbitrary precision decimal numbers	`3.14, 100, 1.0E3`
`integer`	Represents the standard mathematical concept of integer numbers	`1, 10, 100`
`time`	Represents an instance of time in the format HH:MM:SS	`14:12:30`
`date`	Represents a calendar date in the format YYYY-MM-DD	`2001-04-16`

Table 2.1 Some simple data-types in XML Schema

complex type

A complex type refers to the content model of an element that can contain other elements and attributes. A `complexType` element will hold the definition of a complex type. It has an attribute `name` to hold the name of the complex type definition.

Inside a complex type definition, there can be element and attribute declaration, as well as other constrains components such as the `xsd:sequence` element which indicates that the child elements should appear in the order as they are declared.

element

Element is one of the major building blocks in any XML documents. The declaration of an element is an association of its name with a type. An `element` element will be used to declare an XML element, and a `type` attribute can be added to specify the data-type of the element. For example:

```
<xsd:element name="record" type="xsd:string"/>
```

will declare an element named `record` and the element is associated with the data-type `xsd:string`.

attribute

The declaration of an attribute is similar to that of an element. An `attribute` element is used to declare the attribute and a `name` attribute will specify the name of the attribute we are declaring. Optionally we can specify the data-type for the attribute value by using the `type` attribute. For example:

```
<xsd:attribute name="date" type="xsd:date"/>
```

will declare an attribute named `date` with the data-type `xsd:date` as its valid value. To declare an attribute for an element, however, we have to add the complex type declaration to the element. For example:

```
<xsd:element name="record">
    <xsd:complexType>
```

```
        <xsd:attribute name="date" type="xsd:date"/>
        <xsd:string/>
    </xsd:complexType>
</xsd:element>
```

The above example declares an element named `record`, and adds the `date` attribute of `xsd:date` data-type, and the element contains a string value.

2.2.2. Schema Alternatives

Although XML Schema is an “official” development of the W3C, there are some schema alternatives that provide different approaches in modeling XML data structure. Some XML-based application platforms are developed on the basis of particular schema alternatives. XML-Data, XDR (XML-Data Reduced), and Schema for Object-oriented XML (SOX) are examples of such schema alternatives: XDR is the basis for the BizTalk Framework, and SOX is used as the basis for CommerceOne’s e-business initiative.

In our research, we have adopted a tailored schema for our Structured-Element Object Model. The schema binds the XML data with the SEOM and the Java classes that implement the Structured-Element. The schema, however, is embedded in the W3C XML Schema where a primary schema component is added.

2.3 Overview of XPath

The XML Path Language [42] (XPath) is a querying language that is used to address specific parts of an XML data object as nodes within a tree. In support of this, XPath can also handle strings, numbers, and Boolean variables. XPath expressions use a compact non-XML syntax, which allows for easier use within URIs and XML attributes.

This language handles XML data as paths within an abstract hierarchical tree structure of nodes, and a current context node. The addressing capability of XPath also implies the ability to match patterns in the XML data, providing a simple method of querying that data.

XPath can be viewed as a way to navigate around XML documents. It models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes.

The primary syntactic construct in XPath is the expression. An expression is evaluated to yield an object, which has one of the following four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

One important kind of expression is a location path. A location path selects a set of nodes relative to the context node. The result of evaluating an expression that is a location path is the node-set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter sets of nodes.

A location path consists of a sequence of one or more location steps separated by “/”. The steps are composed from left to right and each step in turn selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are unioned together and the final set at the end of the path will be the resulting nodes selected by this location path.

A location step has three parts: an axis, a node test, and zero or more predicates. The XPath axis specifies the direction along which the hierarchy of nodes is being navigated. The node test specifies the node type and expanded-name of the nodes selected along the specified axis. The optional predicate allows further filtering of the node set yielded when the node test has been applied in the specified axis.

There are 13 different axes defined in XPath, including child axis, descendant axis, parent axis, ancestor axis, self axis, attribute axis, and also axes defined for the siblings, preceding or following nodes, etc.

For each location step, there must be a node test. The node test selects a set of nodes from the specified axis by using the qualified name of the node. For nodes that do not have a qualified name, a node type test can be applied. For example, we can use `text()` to select all text nodes, or `comment()` to select all comment nodes. There is also a node test `node()` that will select any node of any type whatsoever.

A predicate is another component in the location step. It filters a node-set with respect to an axis to a new node-set. For each node in the node-set to be filtered, the predication expression is evaluated with that node as the context node, with the number of nodes in the node-set as the context size, and with the proximity position of the node in the node-set with respect to the axis as the context position. The node will be included in the result node set if and only if the predicate expression evaluates to a true value. For a predicate expression, it may yield a boolean value or a number. If the result is a number, the result will be converted to "true" if the number is equal to the context position; and to "false" otherwise.

In our research project, we enhance the query capability when using with SEOM Documents by introducing a query function. The query function is embedded into the predicate term of a location step. This provides a convenient method to select nodes based on the semantics of the element structure.

2.4 Overview of DOM

An XML document is a hierarchical tree of elements and other entities. Document Object Model (DOM) [41] is the standard tree object model used in processing XML documents. The DOM defines an interface for dynamic access and modification of structured data in XML. The interface is platform-independent and language-neutral and is not tied with a specific implementation; it is designed to be used with any programming language and any operating system.

When the DOM is used to manipulate an XML text file, the first thing it does is parse the file, breaking the file out into individual elements, attributes, comments, and so on. The DOM then creates a representation of the XML file as a node tree in the memory, and modification can be made to the tree through the DOM interface. By using the DOM to create and access XML documents, not only the grammar and well-formedness of XML document will be ensured, but it also abstract the content away from the grammar and thus simplifies the document manipulation.

Most of the APIs defined by the W3C DOM specification are interfaces rather than classes. That means that an actual implementation need only expose methods with the defined names and specified operation, not actually implement classes that correspond directly to the interfaces. This allows the DOM APIs to be implemented as a thin veneer on top of legacy applications with their own data structures, or on top of newer applications with different class hierarchies.

The `Node` interface is the primary datatype for the entire Document Object Model. It represents a single node in the document tree. While all objects implementing the `Node` interface expose methods for dealing with children, not all objects implementing the `Node` interface may have children; some types of nodes may have child nodes of various types, and others are leaf nodes that cannot have anything below them in the document structure. Some of the more specialized interfaces are:

- Document
- Element
- Attr
- Text
- CDATASection

The DOM also specifies a `NodeList` interface to handle ordered lists of `Nodes`, such as the children of a `Node`, or the elements returned by the `Element.getElementsByTagName()` method.

In our research project, our Structured-Element Object Model implements the DOM interface, plus an additional interface for SElement type. Details of the modeling will be covered in Section 4.1.

Chapter 3.

Overview of Structured-Element Object Model (SEOM)

XML is a metadata language, which means "data about data."[38] XML data modeling is an activity of using XML to model data [18]. The process may involve finding the most suitable and economical usage of document, elements, attributes and text sections to represent data in the abstraction level [32], which may result in higher reusability and save development effort [23].

In Chapter 1, we have described the need for storing and manipulating complex data objects in XML and the need for extending the DOM interface to address this kind of application. In our research project, we propose a *Structured-Element Object Model* (SEOM), which extends the Document Object Model by introducing a new node type, the *Structured-Element* (SElement).

In this chapter, we will present the overview of our model, including the research object, some general concepts for our model, as well as the approach in implementing the model. This chapter provides the basis for the modeling, processing, and implementation of SEOM, which will be covered in subsequent chapters.

3.1 Introduction

XML is a new standard for representing and exchanging data on the Internet. How to model XML data for web applications is a hot topic that attracts a lot of interests. In modeling XML data, metadata plays one of the most important roles.

XML assigns semantic and structural meanings to the data on the Internet. This makes it possible for web applications and database management systems to manipulate, manage and retrieve the data on the Internet. XML can also be used to define data transfer format, data manipulation algorithm or represent semi-structured data [14].

In general, XML represents semi-structured data with no rigid schema; the basis for processing XML data is to establish a suitable data model for XML data. There has been a lot of work for modeling XML data into relational and object-relational models to facilitate better information management [11, 12, 16]. However, for XML data modeling in application development, an object-oriented data model can reflect the concepts of components in object-oriented programming languages more appropriately; this benefits building of reusable software components [4, 17, 30].

When XML is used in application development, data access inside an XML document is a major consideration. For accessing a record inside a small XML document, it will be rather easy to read all the records to get the one you want; for a larger document, it may be more effective by using some filtering techniques like XPath to reduce the number of records to be processed. However, for an even larger document, the ordinary filtering features (e.g. string matching in XPath) may not be efficient enough, and an indexing structure may be employed to increase the search speed. For an effective indexing, metadata is needed to provide understanding on the contents in the document [18].

Moreover, in application development, it is often that complex data object that represents a single business/logical entity is needed. The single-valued Element object in DOM is not sufficient to carry the complete information. Therefore we need a new data representation that is flexible in representing complex data as a single entity.

In our research, we propose a data model similar to DOM that has a hierarchical tree structure of data objects. A new node type, *Structured-Element* (SElement), is introduced. The node type is given this name because it is similar to the XML Element type but embeds an additional data structure inside. The SElement provides great flexibility in modeling different complex data types. Its internal data structure resembles an indexing structure for the XML data; or the internal data structure itself can stores data in a more efficient manner than in the generic tree structure of DOM. Metadata takes an important part for the system to know what kind of structure should the XML data to bind with. As SElement is the major invention in our model, we name our model Structured-Element Object Model (SEOM).

3.2 Objectives

Our research mainly focuses on using XML to support complex data objects such as "Lists of Lists of Lists", "Trees of Trees of Trees", heterogeneous type like "Trees of Lists of Trees" [25], or other user-defined complex data type that address special needs in any specific fields.

We defined a data model to represent XML documents; the new data model takes care of the meanings in element structures. There are a few objectives for our data model:

- to build reusable software components;
- to define a schema for storing and exchanging XML data containing complex data objects;
- to facilitate marshaling and unmarshaling between XML data and complex data objects;
- to manage complex data objects through a neat interface;
- to hide private data members for a complex data object;
- to be flexible in binding with different complex data types;
- to be extensible in introducing new data types; and

- to be compatible with current hierarchical object models (DOM).

This thesis presents our object modeling for XML data. We define a set of data types and their interfaces. We also define the syntax of SEOM schema, which facilitates declaration of the added *SElement* type specifically. Then we describe how could how are these data objects implemented, and how is the XML data parsed into instance of SElement. Moreover, we also describe two mechanisms for querying *SEOM Document*, i.e. *query wrapper* and *extended XPath query function*. In the following section, we will introduce the general concepts in SEOM.

3.3 General Concepts in SEOM

3.3.1. Data Representation

Data representation is twofold in physical representation and in logical representation. In SEOM, the physical representation means how are data represented in an XML file, and the logical representation means how are the data interfaced to the application programmers [27, 29].

In the conventional Document Object Model, these two representations bind together tightly; there exists a one-to-one mapping for the data types in these two domains. Major node types in DOM, including Element, Attribute and CharacterData, are the same as the data types in XML specification. The simplicity of the model promotes high flexibility in manipulating XML data, however, the lightweight component approach may not be the most suitable and economical representation of complex data.

Consider that many complex data structures like relational data and various indexing trees are frequently used in many programs, storing their contents in XML would bring a lot of convenience and flexibility. However, application programmers are mainly interested in high-level information and contents but not details in indexing or schema modeling. An abstraction for complex data objects would simplify the program design and shorten the development time. Therefore, we will present a XML data modeling in which the physical data representation and the logical data representation are more flexibly coupled.

Physical Data Representation

While a generic tree structure is able to capture the “external” appearance if many different data structures, these structures are indeed different in their meaning and behaviors. A generic tree cannot capture the internal properties of individual “sub-types”. For example, the B-tree, quad-tree, R-tree [15], R*-tree, X-Tree and M-Tree belong to the tree class [11, 19], but each of them defines different ways of indexing a spatial data set and provides different query operations on the data set. In some cases, a structure can be “represented” as a tree, but its fundamental representation is something else. Relational table is a typical case for this kind of structures; the HTML `table` element presents a table-like structure to the users, while it is also a tree-like structure from the viewpoint of Document Object Model; and in Relational DBMS, a table is indexed as B-Tree to optimize various operations, which does not looks like a table anymore.

When designing an XML data representation, the most important structure information is what kind of data structure is allowed. All schema language basically defines the allowed nesting structures of elements. We adopt the XML Schema to declare the element structure for an XML document. Besides “what XML trees are allowed”, another important structure information we raised here is “what is the type of the tree”. Metadata takes the role of specifying this information. The SElement declaration is embedded in a schema when an XML element together with its sub-tree structure should map to a complex data object; the declaration includes how the different components of XML tree (element, attribute, and text nodes) are mapped into the data members in the target class.

Moreover, there are two different approaches in physical data representation:

- First is to embed the complete index-tree structure in XML file, such as a complete R-Tree, or a B-Tree. In this way, the object will be instantiate via a bulk-loading manner.

- Second is to model the data elements only in the XML file, i.e., the data points in R-Tree, or data tuples in a table (instead of the B-Tree indexing structure for table). In this way, the object instantiation will involve a building process for the internal data structure.

Both approaches are valid physical data representations, but they have different advantages (and disadvantages) and require different implementation classes.

Logical Data Representation

While the XML documents can capture a variety of data relationships, the development of large, realistic applications is much simpler by using the most appropriate logical representation. In modeling a large system, only part of the information is frequently accessed, and it is more efficient to make the most desired information as explicit as possible in the representation. This concept is consistent with data encapsulation in an object-oriented programming language.

Currently, there are two major type of logical data representation for XML data:

- one is to focus on reflecting XML data structure in its “natural form”, and
- another type is to reflect business/logical objects;

DOM [41] and JAXB [34] are typical examples of these two types respectively.

These two representation types have their own advantages and disadvantages. The first approach reflects an XML document as a single entity; it provides a generic view to any hierarchical structured document; it is supported by a number of XML-related technologies such as XPath, XSLT, etc. The second approach, on the other hand, reflects each logical entity as a single object; it facilitates better data encapsulation; closely models the object-oriented programming paradigm, and protects against illegal data access in team development.

In our model, we combine the benefits of these two representation types. As a whole, the SEOM resembles the DOM with an additional SElement type and additional functionalities. In addition, SElement is designed to encapsulate logical entities as the data binding technology does. The SElement maps an XML subtree into a single node object, encapsulates the element structure as its internal data and presents a query interface to the programmers. Figure 3.1 shows the structure of an SElement. It is connected to a parent node and a number of child nodes just as an Element would. The only difference is that it exposes a query function that allows programmatic accesses to the internal data structure.

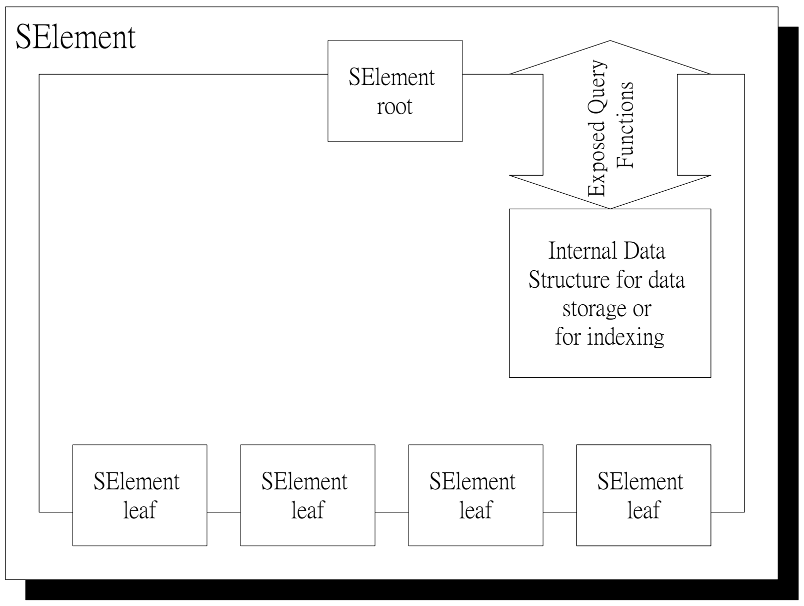


Figure 3.1 Structure of an SElement

3.3.2. Data Binding

In SEOM, we separate the parsed document object from the physical XML file layout; the parsed document object does not reflect the primary XML structure in an one-to-one manner. Schema is used to bind the primary XML constructs to the SElement implementation classes. Figure 3.2 shows the relationship between the data (primary XML constructs), schema, implementation classes, and the SElement object. To make the model generically applicable, we do not define any special tags in the

XML document; instead, we define the bindings in the SEOM Schema. The SEOM Schema has a set of basic constructs; the constructs are generally used for data binding, and they are not tied to any particular model.

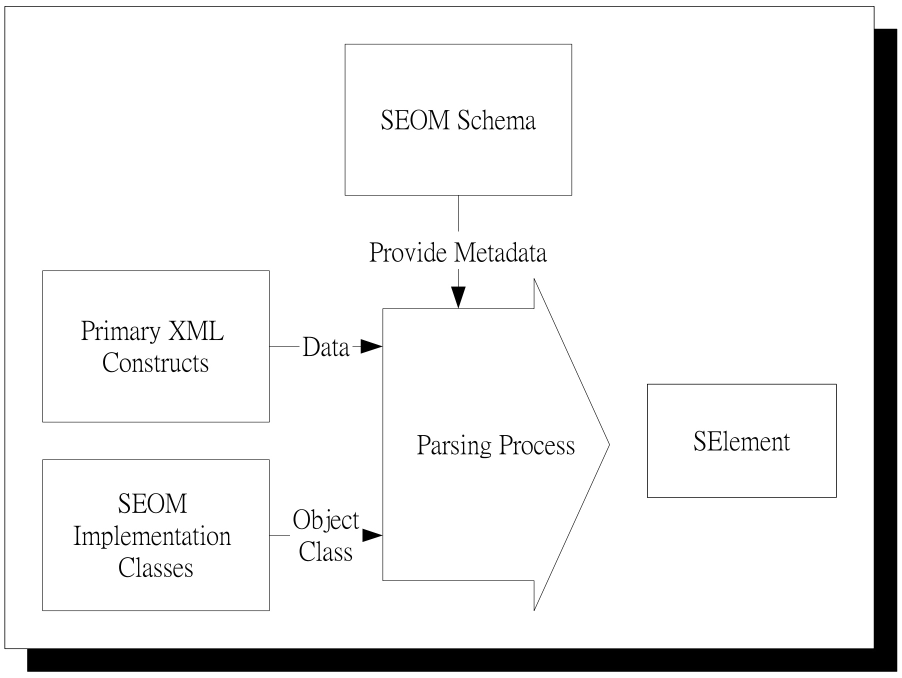


Figure 3.2 Relationship of Data, Schema, Class, and Object Instant

In our research, we combine the techniques in data binding with the tree-like data structure of tree-based parser. Since Java is one of the most popular programming languages for developing Web applications, we use Java objects to represent our proposed objects for XML data [24]. Developers can build applications with Java classes that reflect the logical entities, and to interface those Java classes to the generic DOM structure through the SElement node object. The Java classes are also interface automatically to any XML that conforms to a schema definition. Consequently, a Java class will be loaded with XML data automatically and instantiated as one of the nodes in a larger DOM structure. Details of parsing and data binding will be covered in Section 5.3.

3.3.3. Data Access

Data access is the essential part for a data model. The accessing methods defined will affect the usability of the model.

In accessing an XML document inside a database management system, the query language of that database model is often used. For non-database system, XPath is a technique adopted by the W3C Consortium and it is widely used in Internet applications and client-side scripts to address parts of XML documents.

In this thesis, we define two data accessing methods. First is the *wrapper* method. This method is particularly designed for using in a programming framework; it uses XML messages to wrap a query or a result set. It allows users to retrieve a *query wrapper*, fill in the required parameters, and then submit the query by using the filled wrapper.

Another data access method that can be used in SEOM is the XPath. We extend XPath with an additional query function that can be embedded as a *term* in the *XPath predicate*. This function performs queries similar to the query wrapper, except that a wrapper can take complex type parameter and generate complex type result, which is not possible for the XPath type query.

More details of querying will be cover in section 5.4.

Chapter 4.

SEOM Document Modeling

4.1 Data Modeling

A data model describes the structure, function, and constraints of the data, which focusing on the abstract properties of the data and ignoring the concrete limitations imposed by a computer system. This will be useful to provide a clean concise overview when overwhelmed by the details and implementation.

XML and HTML are similar in a sense that they are formed by inserting a set of tags into a body of text. While HTML tags are pre-defined and allow common browsers to interpret them for drawing the layout of the document. XML tags, on the other hand, are defined by users and are not concerned with display at all. There is no existing software that will automatically understand the tags, unless it is specifically written to interpret the specified structure.

Despite that the semantics of XML document could not be automatically understood, they could still share a common data model with the help of specifications and metadata. The XML specification provides the data types and constraints that would be necessary for a data model. As a markup language, XML itself do not include any explicit operations other that the ones for creating elements in the data type, though the associating standards like DOM and XPath suggest additional operations like the manipulating and querying operations.

There are a number of data models that are relevant to XML, include entity-relational, semantic, graph data model [14] and etc. In the following subsections, we will first introduce a simple XML data models, and followed by the data model defined in the Structured-Element Object Model.

4.1.1. Simple XML Data Model

This section introduces a simple data model for XML. The model illustrates the fundamentals of a XML data model, like creating documents, elements, attributes, and character data regions. The simplicity will help you to have a brief idea of XML data modeling before we introduce our data modeling in SEOM.

Type

The data types of the data model are:

Data Type	**Definition**
`Document`	A named entity with one (root) Element
`Element`	An entity with a type name, a collection of attributes, and an ordered collection shared by character data and elements.
`Attribute`	A name-value pair. Both the name and the value are strings.
`Character Data`	A string value.

Table 4.1 Types in simple XML data model

The union of element, and character data type are called `Node`, which forms the nodes in an element tree.

For collection types, we have `AttributeSet` for a collection of attributes; and `NodeList` to denote an ordered collection of nodes.

Operations

In the simple XML data model, operations defined includes add, delete, and retrieve data, while querying is not included. The convention of describing an operation will be:

```
<result type> <operation name> (<type1> <parameter name1>, …)
```

The convention is similar to that of describing methods in the Java programming language. Following are the definition of operations for different data types.

`Document`	Document `newDocument`(String `name`) *Create a new document with unique* `<name>`
	Element `createDocumentElement`(Document `document`, String `tag`) *Create the (root) document element for* `<document>` *with tag name* `<tag>`*, return the new element*
	Element `getDocumentElement`(Document `document`) *Get (root) document element for* `<document>`
	String `getName`(Document `document`) *Get the name of* `<document>`

Table 4.2 Operations for Document in simple XML data model

`Element`	String `getTag`(Element `element`) *Get the tag (element type) name for* `<element>`
	Attribute `createAttribute`(Element `element`, String `name`, String `value`) *Add an attribute to an* `<element>` *with* `<name>` *and* `<value>`*, return the new attribute*
	CharData `createCharData`(Element `element`, String `data`) *Add a character data region to the end of an* `<element>` *consisting of a string* `<data>`*, return the new CharData*
	CharData `createCharData`(Element `element`, String `data`, Integer `index`)

	Add a character data region to `<element>` *consisting of a string* `<data>` *at index location* `<index>`*, return the new CharData*
	Element `createElement`(Element `element`, String `tag`) *Add a subelement to the end of an* `<element>` *with element type name* `<tag>`*, return the new subelement*
	Element `createElement`(Element `element`, String `tag`, Integer `index`) *Add a subelement to the end of an* `<element>` *with element type name* `<tag>` *at index location* `<index>`*, return the new subelement*
	Attribute `getAttribute`(Element `element`, String `name`) *Get the attribute of* `<element>` *with name* `<name>`
	Node `getChild`(Element `element`, Integer `index`) *Get the child of* `<element>` *at* `<index>`
	Boolean `removeAttribute`(Element `element`, String `name`) *Remove the attribute of* `<element>` *with name* `<name>`*, return true if the attribute existed*
	Boolean `removeChild`(Element `element`, Integer `index`) *Remove the child of* `<element>` *at* `<index>`*, return true if the child existed*
	AttributeSet `getAttributes`(Element `element`) *Get all the attributes of* `<element>` *to an attribute set collection*
	NodeList `getChildren`(Element `element`) *Get all the children of* `<element>` *to an node list collection*

Table 4.3 Operations for Element in simple XML data model

`Attribute`	String `getName`(Attribute `attribute`) *Get the name of* `<attribute>`
	String `getValue`(Attribute `attribute`) *Get the value of* `<attribute>`

	String `setValue`(Attribute `attribute`, String `value`) *Set the value of* `<attribute>` *to* `<value>`

Table 4.4 Operations for Attribute

`Character data`	String `getData`(CharData `chardata`) *Get the data string of* `<chardata>`
	String `setData`(CharData `chardata`, String `data`) *Set the data value of* `<chardata>` *to be* `<data>`

Table 4.5 Operations for Character Data in simple XML data model

`Node`	String `getType`(Node `node`) *Get the type of* `<node>` *as a string, either* `'Element'` *or* `'CharData'`

Table 4.6 Operation for Node in simple XML data model

`NodeList`	Integer `getLength`(NodeList `nodeList`) *Return number of nodes in* `<nodeList>`
	Node `item`(NodeList `nodeList`, Integer `index`) *Retrieve from* `<nodeList>` *the node at location* `<index>`

Table 4.7 Operations for NodeList in simple XML data model

`AttributeSet`	Integer `getLength`(AttributeSet `attributeSet`) *Return number of attributes in* `<attributeSet>`
	Attribute `item`(AttributeSet `attributeSet`, Integer `index`) *Retrieve from* `<attributeSet>` *the attribute at location*

	`<index>`

Table 4.8 Operations for AttributeSet in simple XML data model

The simple XML data model provides the basic operations of manipulating XML documents, which will be the framework of more complicated data models.

Constraints

The constraints for the simple data model are:

- Document names are unique
- The document element has no parent, and all other elements have an element node as a parent which will form a tree structure.
- No attribute name may appear more than once in an element.

4.1.2. SEOM Data Model

Based on the W3C DOM and XML Specification, we develop the Structured-Element Object Model with enhanced querying for complex data structures and better usability in manipulating the structure.

It is often that programmers bind XML documents to DOM and manipulate the document through the tree-like data structure in DOM, in which the XML elements, attributes, and text sections in the XML document are mapped into DOM objects in the tree. A DOM tree is a collection of nodes where each node has at most one parent node and may have any number of ordered child nodes. The `Nodes` are objects defined in the W3C DOM Specification, which includes `Attribute`, `CharacterData`, `Comment`, `Document`, `Element`, and `Node`. Besides the linkage to parent nodes and child nodes, the objects often embedded simple name-value pair of string type.

In SEOM, there is an added data type named *Structured-Element* (abbreviated as SElement). The `SElement` contains an internal data structure, which allows queries made to the `SElement` and generates result as `NodeList`.

Types

The types are based upon those of the previous section. The W3C XML Specification and the W3C DOM Specification are also considered, however, some elements such as the `Comment` type and the `Processing Instruction` type are insignificant in our modeling and thus they are not included here.

Data Type	Definition
`Document`	A `Document` represents the XML Document and contains one `Element` which is called *RootNode*..
`Element`	Except the tags declared as `SElement` in the Schema, most tagged nodes of the XML document will become `Element` nodes. Each `Element` node has a name same as the tag, one parent node of `Element` or `SElement` type except the *RootNode*, and optionally a collection of `Attribute` nodes, and an ordered collection of `Element/SElement/CharData` nodes as children.
`SElement`	Tagged nodes of XML document which have binding declaration in the Schema will be created as `SElements`. Each `SElement` node has a name same as the tag of the *root-tag* of the XML segment and a type name declared in the corresponding *class declaration* in the Schema. It has one parent node of `Element` or `SElement` type, and a set of `Attribute` nodes with declaration in Schema, and an ordered collection of `Element/SElement/CharData` nodes as children. The binding-class of `SElement` will defines an internal data structure for `SElement`, with corresponding query operations that maps to a sub-set of its child nodes. The list of implemented query operations is available from the `SElement`.
`CharData`	The string between the tags becomes `CharData` node. Only the full-length strings are considered to form the

	nodes, sub-strings of a longer (parent) string will not form `CharData` node. The XML Specification also defines a CDATA section that is a type of `CharData` node where the text strings are quoted.
`Node`	A `Node` refers to an `Element`, a `SElement`, or a `CharData` node. It is the superclass of these data types and provides access to the element tree through parent/child and sibling relationships.
`NodeList`	A `NodeList` consists of an ordered collection of `Nodes`. Each `Node` in the list is associated with an integer index.
`Attribute`	The name-value pairs in the tags form `Attribute` nodes. One name-value pair will form one `Attribute` node. The values are simple strings, and multi-valued attribute are stored as a white-space delimited string.
`AttributeSet`	It represents a set of `Attributes` under an `Element/SElement`.

Table 4.9 Types in SEOM

In addition to the simple XML data model, we make the `Node` data type explicit as in the DOM Specification to make it a super-class for the `Element` nodes and `CharData` nodes. The `Node` object allows tree transverse with simple programming interface. The collection data types, `NodeList` and `AttributeSet`, are also made explicit and they provides convenient operations when a group of objects are involved.

The most significant change of the model is the addition of the new data type, `SElement`, which combines data structure and processing logic and put them into a single object. It provides concise interface to the programmers when dealing with such kind of data structures and take off the unnecessary burdens.

Operations

Suggested from the W3C DOM Specification, the constituents of a document are associated with the document instead of an element. Thus the creation of various node-types are operations of `Document`; and the operation for appending them to the tree is operation of the `Node` type; and `Attribute` type is created from the `Document` and appended by operation of `Element` type.

All Node Types	String `asXML`(Node `node`) *To generate the text with XML markups corresponding to the document under* `<node>`
	String `asText`(Node `node`) *To generate the text without markups corresponding to the document under* `<node>`

Table 4.10 Generic Operations in SEOM

`Document`	Document `newDocument`(String `name`) *Create a new document with unique* `<name>`
	String `getName`(Document `document`) *Get the name of* `<document>`
	Void `setDocumentElement`(Document `document`, Element `element`) *Set the (root) document element for* `<document>` *be to* `<element>`
	Element `getDocumentElement`(Document `document`) *Get (root) document element for* `<document>`
	Element `createElement`(Document `document`, String `tag`) *Create an Element in* `<document>` *with element type name* `<tag>`*, return the new element*
	SElement `createSElement`(Document `document`, String `tag`, Type `schema`) *Create an SElement in* `<document>` *with element type name* `<tag>`*, and associate with the type as defined in* `<schema>`. `<schema>` *is a DOM document with predefined format to give information on the internal data structure of the SElement*

	CharData `createCharData`(Document `document`, String `value`) *Create a CharData node in* `<document>` *consisting of a string* `<data>`*, return the new CharData*
	Attribute `createAttribute`(Document `document`, String `name`, String `value`) *Create an attribute in* `<document>` *with* `<name>` *and* `<value>`*, return the new attribute*
	NodeList `getElementsByTagname`(Document `document`, String `tag`) *Retrieve all elements in* `<document>` *with element type name* `<tag>`
	Element `query`(Document `document`, Element `querywrapper`) *Submit a query wrapped in* `<querywrapper>` *to* `<docment>`*, results are wrapped in Element*
	Document `transform`(Document `document`, Document `stylesheet`) *The operation takes an XSLT stylesheet* `<stylesheet>` *as argument, apply the transformation on* `<document>` *as specified in W3C XSLT Specification, and return the transformed Document as result*

Table 4.11 Operations for Document in SEOM

`Element`	String `getTagName`(Element `element`) *Get the tag name of* `<element>`
	Attribute `setAttribute`(Element `element`, Attribute `attribute`) *Add the node* `<attribute>` *to* `<element>`
	Attribute `getAttribute`(Element `element`, String `name`) *Get the attribute of* `<element>` *with name* `<name>`
	Boolean `removeAttribute`(Element `element`, String `name`) *Remove the attribute of* `<element>` *with name* `<name>`*, return true if the attribute existed*

	AttributeSet `getAttributes`(Element `element`) *Get all the attributes of* `<element>`
	Node `appendChild` (Element `element`, Node `node`) *Append* `<node>` *as child of* `<element>`*, return the subelement*
	Node `appendChild`(Element `element`, Node `node`, Integer `index`) *Append* `<node>` *as child of* `<element>` *at index location* `<index>`*, return the subelement*
	NodeList `getChildren`(Element `element`) *Get the children of* `<element>` *in order*
	Boolean `removeChild`(Element `element`, Node `child`) *Remove the node* `<child>` *from* `<element>`*, return true if the child existed*
	NodeList `path`(Element `element`, String `xpath`), or Boolean `path`(Element `element`, String `xpath`), or Number `path`(Element `element`, String `xpath`) *Takes a string* `<xpath>` *as parameter and act on* `<element>` *according to the XPath Specification, return the result as* `NodeList`, `Boolean` *or* `Number` *depending on the function used in* `<xpath>`
	NodeList `getElementsByTagname`(Element `element`, String `tag`) *Retrieve all elements in* `<element>` *with element type name* `<tag>`

Table 4.12 Operations for Element in SEOM

The SElement is a major creation with respect to the current modeling in DOM. Details of this data type will be covered in later section.

`SElement`	String `getTagName`(SElement `selement`) *Get the tag name of* `<selement>`
	String `getTypeName`(SElement `selement`)

Get the type name of `<selement>`
Document `getSchema`(SElement `selement`) *Get the schema document of* `<selement>`
Attribute `setAttribute`(SElement `selement`, Attribute `attribute`) *Add the node* `<attribute>` *to* `<selement>`
Attribute `getAttribute`(SElement `selement`, String `name`) *Get the attribute of* `<selement>` *with name* `<name>`
Boolean `removeAttribute`(SElement `selement`, String `name`) *Remove the attribute of* `<selement>` *with name* `<name>`*, return true if the attribute existed and is an optional attribute*
AttributeSet `getAttributes`(SElement `selement`) *Get all the attributes of* `<selement>`
Node `appendChild` (SElement `selement`, Node `node`) *Append* `<node>` *as child of* `<selement>`
Element `addSChild`(SElement `selement`, Element `datawrapper`) *Add a record into* `<selement>`*, where* `<datawrapper>` *is an Element of specific format to encapsulate the data as well as needed information*
Boolean `removeChild`(SElement `selement`, Node `child`) *Remove the node* `<child>` *from* `<selement>`*, return true if the child existed and can be removed freely*
NodeList `getChildren`(SElement `selement`) *Get the children of* `<selement>` *in order*
NodeList `queryMethods`(SElement `selement`) *Returns the queryMethods available in* `<selement>`*, wrapped in form of* `Elements`
NodeList `query`(SElement `selement`, Element `querywrapper`) *Submit a query wrapped in* `<querywrapper>` *to* `<selement>`*, results are wrapped in Elements and the collection is resulted as*

	`NodeList`
	NodeList `path`(SElement `selement`, String `xpath`), or Boolean `path`(SElement `selement`, String `xpath`), or Number `path`(SElement `selement`, String `xpath`) *Takes a string* `<xpath>` *as parameter and act on* `<selement>` *according to the XPath Specification, return the result as* `NodeList`, `Boolean` *or* `Number` *depending on the function used in* `<xpath>`
	NodeList `getElementsByTagname`(Element `selement`, String `tag`) *Retrieve all elements in* `<selement>` *with element type name* `<tag>`

Table 4.13 Operations for SElement in SEOM

`Attribute`	String `getName`(Attribute `attribute`) *Get the name of* `<attribute>`
	String `getValue`(Attribute `attribute`) *Get the value of* `<attribute>`
	String `setValue`(Attribute `attribute`, String `value`) *Set the value of* `<attribute>` *to* `<value>`

Table 4.14 Operations for Attribute in SEOM

`CharData`	String `getData`(CharData `charData`) *Get the data string of* `<charData>`
	String `setData`(CharData `charData`, String `data`) *Set the data value of* `<charData>` *to be* `<data>`

Table 4.15 Operations for CharData in SEOM

As in the W3C DOM Specification, the Node type here is more important then in the simple XML data model. The Node structure forms the main body of document tree. The operations in the Node type allow a tree of nodes to be created.

`Node`	String `getType`(Node `node`) *Get the type of* `<node>` *as a string, possible values are "SElement", "Element", or "CharData"*
	Document `getOwnerDocument`(Node `node`) *Return the document to which* `<node>` *belongs*
	Element `getParentNode`(Node `node`) *Get the parent element of* `<node>`
	Node get `PreviousSibling`(Node `node`) *Get the previous sibling of* `<node>`
	Node `getNextSibling`(Node `node`) *Get the next sibling of* `<node>`

Table 4.16 Operations for Node in SEOM

`NodeList`	Integer `getLength`(NodeList `nodeList`) *Return number of nodes in* `<nodeList>`
	Node `item`(NodeList `nodeList`, Integer `index`) *Retrieve from* `<nodeList>` *the node at location* `<index>`

Table 4.17 Operations for NodeList in SEOM

`AttributeSet`	Integer `getLength`(AttributeSet `attributeSet`) *Return number of attributes in* `<attributeSet>`
	Attribute `item`(AttributeSet `attributeSet`, Integer `index`) *Retrieve from* `<attributeSet>` *the attribute at location* `<index>`

Table 4.18 Operations for AttributeSet in SEOM

Constraints

The constraints in this data model will assure consistency of data as well as the manipulation. Several constraints are also adopted from the W3C XML Specification and the W3C DOM Specification. The constraints are:

- Each document have a unique name.
- All elements other than the root element have exactly one element node as a parent. The root element has no parent.
- No attribute name may appear more than once in an element.
- The integer index is a set of consecutive integers beginning from 0.
- Number of previous sibling may be 0 or 1.
- Number of next sibling may be 0 or 1.

4.2 Schema Modeling

A valuable XML concept is the ability to define your own XML vocabulary. An XML vocabulary is an industry-specific XML information model or document type that you define for XML data sharing. Applications use a constrained document type as the basis in processing.

XML Schema provides metadata for describing grammar of an XML document by defining the elements used in the document. However, the syntactic structure, by itself, can only facilitate document validating purpose and nothing more. The programmers still need to know what to do with each schema-defined structure and write codes to process them.

In simplest word, "metadata" means "data about data". It should not be restricted to grammar checking. Metadata for data-structure modeling is one of the valuable functions that can be associated with XML Schema to provide a basis for querying

XML documents. In the absence of a schema we could discover the general structure of a class of XML documents by examining a number of such documents. On the other hand, with an XML schema we can assure the permitted structure of the document elements and therefore can design queries specific to those structures.

The issue for data-structure modeling brings significant benefits in building XML applications too. Current technologies interface XML documents at the level of XML structure and we navigate the documents by interfaces such as DOM and XPath which refer to the "explicit structure" of the XML documents. Developers or users are expected to re-discover the meaning of the XML structures, which is a waste of programming efforts. We need to develop tools so that the developer can work at a higher-level of abstraction to free them from programming routines to transform XML structures to other "useful" data structures before the data can be queried.

4.2.1. SEOM Schema

The W3C XML Schema is one of the most widely adopted schema technology among various XML parties. We would take it as the ground and embed our SEOM-extended schema with different namespace in a valid XML Schema document.

Our extended schema has the namespace URI of `http://www.cse.cuhk.edu.hk/~ckma1/SEOM`. We will use the namespace prefix of `seom` to refer to that URI and it will be used throughout this document.

seom:selement

The `SElement` is the only new data type derived from the DOM data model. Therefore, the SEOM Schema extends the XML Schema only in defining XML structures that relates to `SElement`, which includes the attributes defined under a SElement.

As SElement is extending the Element data type, it will be useful to begin with the

Element declaration in XML Schema. The simplest element type declaration is when an element contains only a primitive data type. For example:

```
<xsd:element name="myElement" type="xsd:string"/>
```

is the element type declaration for a element with type name `myElement` and contains one `xsd:string` as its child.

For a more complex element structure, the structure is declared between a pair of `<xsd:complexType></xsd:complexType>` tags, where multiple of elements and attributes can be declared, and various constraints can be applied on the child nodes.

Now we will see the declaration of a `SElement`. Since SElement type will not contain `simpleType` as in Element, it is not necessary to distinguish `simpleType` and `complexType`. The root of a `SElement` declaration is:

```
<seom:selement name="mySElement" class="model"/>
```

The example defines a `SElement` with type name `mySElement`, which will bind to the class `model`, the name `mySElement` is used as the tag in the XML document and also used as a reference point in other part of the Schema. In next chapter we will discuss issues in data binding and how is the class `model` related to the Schema and XML document.

seom:rootNode / seom:internalNode / seom:leafNode

Within the `SElement` declaration, there will be three kinds of sections, named `rootNode`, `internalNode`, and `leafNode`, which correspond to the group of XML nodes that will be mapped into the same `SElement` object. The `rootNode` section can only exists once and does not have any attributes, the remaining two kinds of sections can exist in multiple, each of them have an attribute `id` for being referenced in other parts in the current `SElement` declaration and an `name` attribute which mapped to the tag name of XML elements just like that of the ordinary element data type. The following example declares a `SElement` bind to the class `model`, which mapped to a sub-tree in the XML document with `<head>` as the root of the sub-tree, `<node>` as the body of the sub-tree, and `<leaf>` as the leaf nodes of the sub-tree. Child nodes under `<leaf>` are also child nodes for the `SElement` instance.

```
<seom:element name="head" class="model">
    <seom:rootNode>
    ...
    </seom:rootNode>

    <seom:internalNode id="node1" name="node">
    ...
    </seom:internalNode>

    <seom:leafNode id="leaf1" name="leaf">
    ...
    </seom:leafNode>
</seom:element>
```

In each of these sections, there will be declaration of attributes, parameters, and the element structure under the node.

seom:attribute

The "SEOM parameter" is not a type actually, but is a variation of attribute. This is because the content of each declared instance of element may be bind to the target model and processed using a different set of parameters. The schema should allow the schema author to specify the parameters in schema or to provide a linkage from schema to the XML document instance where the parameters will be given.

The declaration of attribute/parameter share the same tag `<seom:attribute>`. Both of them will have an attribute `para` where the value is model-specific and mapped to internal variables by the model class. Besides the `para` attribute, there will be a `value` attribute for parameters types, or a `name` attribute for attributes. For example:

```
<seom:attribute para="ID" value="1"/>
```

represents a parameter `ID` for the model and will pass "1" as the parameter value when the SElement is built; while

```
<seom:attribute para="ID" name="id"/>
```

will map the parameter `ID` to attribute `id` in the XML document under the corresponding object.

For the declaration of attribute type, the `<xsd:simpleType>` can be used within `<seom:attribute>` to add restrictions to permitted values of attributes.

seom:value

The `<seom:value>` section declares the valid element structure under a node, which is essentially the same as the `<xsd:complexType>` tag. When referencing to `<seom:rootNode>`, `<seom:internalNode>` or `<seom:leafNode>`, cardinality can be defined by `minOccurs` and `maxOccurs` attribute with integer values > 0 or with the keyword `"unbounded"` (for `maxOccurs`). Moreover, data types in XML Schema can be included as child nodes and statements like `<xsd:restriction>` or `<xsd:sequence>` can be applied, but a few constraints are added:

- `<seom:value>` for `<seom:rootNode>` type should at least references to one `<seom:internalNode>` or `<seom:leafNode>` type.

- `<seom:value>` for `<seom:internalNode>` type should at least references to wither one `<seom:internalNode>` type or one `<seom:leafNode>` type.
- Any `<seom:internalNode>` should references to `<seom:leafNode>` type directly or indirectly (through multiple-levels of other `<seom:internalNode>`).
- `<seom:leafNode>` could not reference to `<seom:internalNode>` or `<seom:leafNode>`.

These constraints are added to ensure the tree structure of XML document is not being violated.

4.2.2. Creating a Schema

To put the things together, we will create a schema and accompany with explanation in a step-by-step manner for easier understanding.

The basic skeleton for an XML Schema is the `<xsd:schema>` element, with its associated namespace declaration; the additional namespace `seom` is for the extended schema.

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

   xmlns:seom="http://www.cse.cuhk.edu.hk/~ckma1/SEOM">
</xsd:schema>
```

The schema document itself is a well-formed XML document. First, we add an ordinary `doc` element which contains a `mySElement` SElement using a reference pointer.

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:seom="http://www.cse.cuhk.edu.hk/~ckma1/SEOM">
    <xsd:element name="doc">
        <seom:selement ref="mySElement"/>
    </xsd:element>
</xsd:schema>
```

Following shows the skeleton of mySElement with SEmodel type where the details of its sections hidden:

```
<seom:selement name="mySElement" class=”SEmodel”>
    <seom:rootNode/>
    <seom:internalNode id="node1" name="node"/>
    <seom:leafNode id="leaf1" name="leaf"/>
</seom:selement>
```

In the <seom:rootNode> element, we define a parameter BRANCH_FACTOR with value equals to "4", and declare that there will be at most 4 <seom:internalNode> under a rootNode:

```
<seom:rootNode>
    <seom:attribute para="BRANCH_FACTOR" value="4"/>
    <seom:value>
        <seom:internalNode ref="node1" maxOccurs="4"/>
    </seom:value>
</seom:rootNode>
```

Then we define the <seom:internalNode> element, which associate the model-specific parameters ID, X1, X2, Y1, and Y2 to XML document attributes id, x1, x2, y1, and y2 correspondingly. The <xsd:choice> element specify that either <seom:internalNode> will occur or <seom:leafNode> will occur.

```
<seom:internalNode id="node1" name="node">
    <seom:attribute para="ID" name="id"/>
    <seom:attribute para="X1" name="x1"/>
    <seom:attribute para="X2" name="x2"/>
    <seom:attribute para="Y1" name="y1"/>
    <seom:attribute para="Y2" name="y2"/>
    <seom:value>
        <xsd:choice>
            <seom:internalNode ref="node1" maxOccurs="4"/>
            <seom:leafNode ref="leaf1" maxOccurs="4"/>
        </xsd:choice>
    </seom:value>
</seom:internalNode>
```

Finally, we define the `<seom:leafNode>` element which stores a single string as the information:

```
<seom:leafNode id="leaf1" name="leaf">
    <seom:attribute para="X" name="x"/>
    <seom:attribute para="Y" name="y"/>
    <seom:value>
        <xsd:string/>
    </seom:value>
</seom:leafNode>
```

Following is the complete Schema document:

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

    xmlns:seom="http://www.cse.cuhk.edu.hk/~ckma1/SEOM">
    <xsd:element name="doc">
        <seom:selement ref="mySElement"/>
```

```
</xsd:element>

<seom:selement name="mySElement" class=”SEmodel”>
    <seom:rootNode>
        <seom:attribute para="BRANCH_FACTOR" value="4"/>
        <seom:value>
            <seom:internalNode ref="node1" maxOccurs="4"/>
        </seom:value>
    </seom:rootNode>

    <seom:internalNode id="node1" name="node">
        <seom:attribute para="ID" name="id"/>
        <seom:attribute para="X1" name="x1"/>
        <seom:attribute para="X2" name="x2"/>
        <seom:attribute para="Y1" name="y1"/>
        <seom:attribute para="Y2" name="y2"/>
        <seom:value>
            <xsd:choice>
                <seom:internalNode
                    ref="node1" maxOccurs="4"/>
                <seom:leafNode ref="leaf1" maxOccurs="4"/>
            </xsd:choice>
        </seom:value>
    </seom:internalNode>

    <seom:leafNode id="leaf1" name="leaf">
        <seom:attribute para="X" name="x"/>
        <seom:attribute para="Y" name="y"/>
        <seom:value>
            <xsd:string/>
        </seom:value>
    </seom:leafNode>
    </seom:selement>

</xsd:schema>
```

Figure 4.1 An Example of SEOM Schema

Chapter 5.

SEOM Document Processing

5.1 SEOM Document Processing

XML language, by itself, is just a language to define markups in a document and do nothing else. To manipulate the data in a useful manner, we write programs to process the documents. This involves binding the XML data into data types in a programming language. As object-oriented programming languages are good at defining new data types, it will be easier to defined data types to match the XML data model, and thus they are suitable for implementing the XML applications. Java, for its good support in XML, is chosen in our research project for implementing the SEOM Document and Element types.

In Chapter 4, we have cover the data model and schema definition for SEOM data types. In this chapter, we will describe the processes associated with SEOM data types. Firstly, we will give details of the classes that made up the SEOM Document and Element types. Then we will describe the process of parsing XML document into SEMO Document, and finally, we will also cover the querying of SEMO Document.

5.2 The Classes

For an SEOM Document instance, it consists of five types of classes:

- Classes from the DOM API;
- SEOM Document class;

- Abstract SElement class;
- Generic SElement class; and
- Implement SElement classes for various structures.

5.2.1. SEOM Document Class

An SEOM Document corresponds to an XML document. It resembles the DOM Document with additional interfaces for the added features in SEOM. Similar to a DOM document, the SEOM document is an in-memory representation of XML documents. Both of they are based on a tree-like structure to model the XML document structure, and they contains the same set of nodes, except that an SEOM document has added methods to reflect the SEOM-defined data type, SElement. Added or changed methods include the constructor method and the query methods. And example of SEOM Document is shown in Figure 5.1, which shows that the document tree node consists of both Element and SElement types.

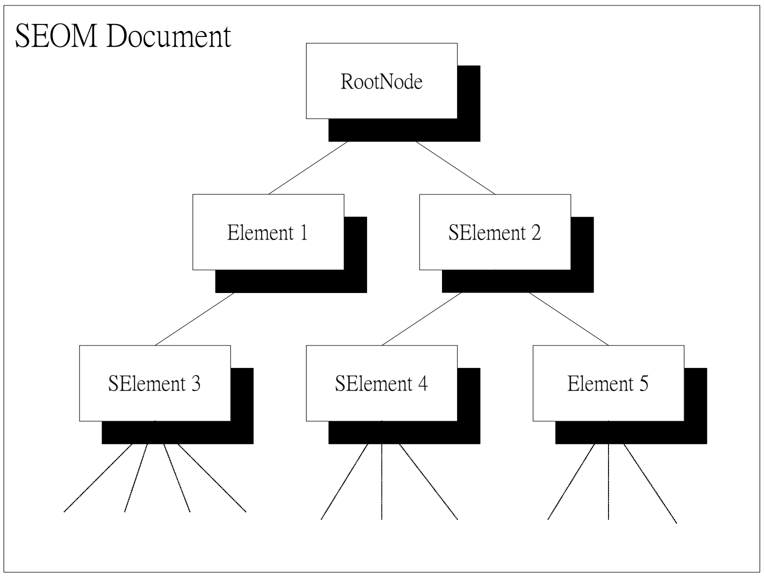


Figure 5.1 An Example of SEOM Document

Constructor Method

Similar to the DOM Document, the SEOM Document has one child node called the root node, which provides a starting point for accessing the data in the document. An SEOM Document is converted from a DOM Document instance. Its constructor method takes a DOM Document (called the *data element*) and an XML Schema (called the *schema element*) as parameters. The constructor method will first parse the schema element to find the namespace associated with SEOM, and then retrieve the associated schema declarations.

The constructor method will get all the Elements with the type name same as the value of *name* attribute in the Schema. Then the Elements will be validated against the Schema, and the process will stop if the element structure does not follow the Schema.

For the matching Elements, they will be transformed to SElements – the generic SElement constructor will be invoked with the Elements, one in a time, and the corresponding schema element as parameters. The generic SElement constructor will return a SElement instance, or throw out exceptions on failures.

When an SElement instance is returned, it will replace the original Element in the original position, whereas the original Element will be detached and threw away.

Query Method

The SEOM Document implements three query operations:

- The `DOM()` operation will retrieve all direct children of a target node, which offers a basic navigation controls to all Element/SElement nodes in the Document.
- The `Data()` operation will retrieve the sub-tree of a target node in XML form, and is particularly useful for retrieving the character data under an Element/SElement node since that would not be accessible through other query methods.
- The `query()` operation is a generic interface that accepts user query. The query may be resolved to the previous two query operations, or pass to the `query()` operation or `queryMethod()` operation in a target SElement.

The `query()` operation (refer to Table 4.11) is the only query interface accessible through the API. The method takes a DOM Element as parameter, called *query wrapper*, which wraps the query details. The wrapper have a root element `<query></query>` with an attribute `path` which points to the target node using XPath. The path is represented in the unique XPath expression from the document root which will return a node set of one node. The unique XPath expression uses the XPath index operator to restrict the path if multiple elements with the same name occur on the path.

Another attribute for the wrapper's root element is `queryMethod`, which corresponds to the method to be invoked on the target Element. When this attribute is absent or is empty-valued, such as:

```
<query path="/root/myRtree"/>
```

the `queryMethod()` method on the target will be invoked, which will retrieve a list of available query methods from the target. On the other hand, when the attribute `queryMethod` is non-empty, the `query()` method on the target will be invoked instead, and the wrapper will be passed into the method call as parameter. A list of `Element` will be returned. Each Element wraps a result of the query.

However, there are a few exception cases needed special handling. First is when the `queryMethod` attribute points to one of the two special function that implemented on the Document level; and second is when the path points to an Element node but not an SElement node.

For the first case, i.e. the value for `queryMethod` is “DOM” or “Data”, the Document's query operation, `DOM()` and `Data()`, will be invoked and the target node will not be queried. These methods are same for both Element and SElement objects.

For the second case, as there is no query operations defined for an Element object, unless the `queryMethod` points to “DOM” or “Data” (which will be intercepted and handled by the Document), an error message will be generated and returned.

5.2.2. Abstract SElement Class

Although the SElement data type is defined as a single data type in section 4.1.2, it is modeled by a number of classes in Java implementation which includes the abstract class for SElement, the default/generic class for the type, as well as the classes that actually implement various internal data structures.

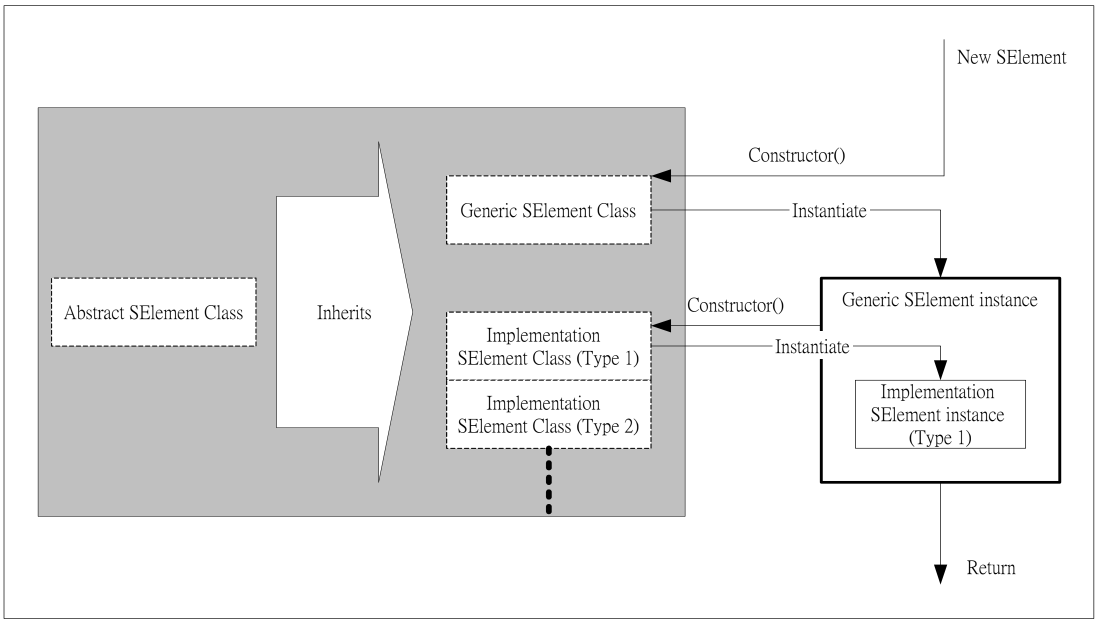

New SElement
Constructor()
Generic SElement Class
Instantiate
Abstract SElement Class
Inherits
Constructor()
Generic SElement instance
Implementation SElement Class (Type 1)
Instantiate
Implementation SElement instance (Type 1)
Implementation SElement Class (Type 2)
Return

Figure 5.2 Creation of SElement and Relation Between Different SElement Classes

The *Abstract SElement* class is the abstract superclass for all SElement data types and can not be used to instantiate any SElement object. It extends the DOM Element class to inherit the methods of DOM Element. It also defines abstract methods that all subtypes of SElement have to implement. The abstract methods include constructor method, `query()` method, and `queryMethod()` method. Operations for SElement are specified in Table 4.13.

The relation between Abstract SElement class, Generic SElement class, Implementation SElement classes and their corresponding instance is shown in Figure 5.2.

5.2.3. Generic SElement Class

The *Generic SElement* class inherits the abstract class and implements the class constructor, the `query()` method and the `queryMethod()` method. It is the only SElement class accessible to programmers. It is used to instantiate the SElement objects in a SEOM Document instance.

The Generic SElement class is a wrapper class for implementation SElement classes that actually implement the internal data structure to hold the data. While the `query()` method and the `queryMethod()` method actually links the call to the member SElement object directly and do nothing else, however, the constructor method of Generic SElement class is responsible in identifying the actually class needed by the structure type specified and load the corresponding class dynamically.

Its constructor method takes a data element and a schema element as the parameters. The schema element corresponds to the SElement declaration in a SEOM Schema. It specifies the structure type of the SElement and the parameters needed for the specific SElement type. The data element is the XML Element that matches the `name` attribute in the SElement schema declaration, which will then be wrapped into an SElement instance. The constructor method will try to fetch the Java class that implements the structure type, and, on success, initiate the *Implementation SElement* with the same parameters that the constructor method had received. The object returned from the constructor method of the Implementation SElement class will be wrapped as a member object of the Generic SElement object, and the Generic SElement object will be returned to the caller.

With the Generic SElement class, XML application programmers can create SElement instance by simply invoke the constructor element with data element and the schema element without worrying about the structure type of data. This simplifies the task for creating SElements.

5.2.4. Implementation SElement Classes

The *Implementation SElement* classes are the group of classes. An Implementation SElement embeds a data structure to hold the data, and implements query operations for that particular data structure. The structure for an Implementation SElement is shown in Figure 5.3. These classes implements interface of the Abstract SElement class. The three main operations for the SElement types are:

- constructor method;
- `queryMethod()` method; and
- `query()` method.

We have implemented an *Implementation SElement classes*, the *RTree*, in our web-based XML query system, and more structure types can be added progressively. Details of these two classes will be covered in Chapter 6.

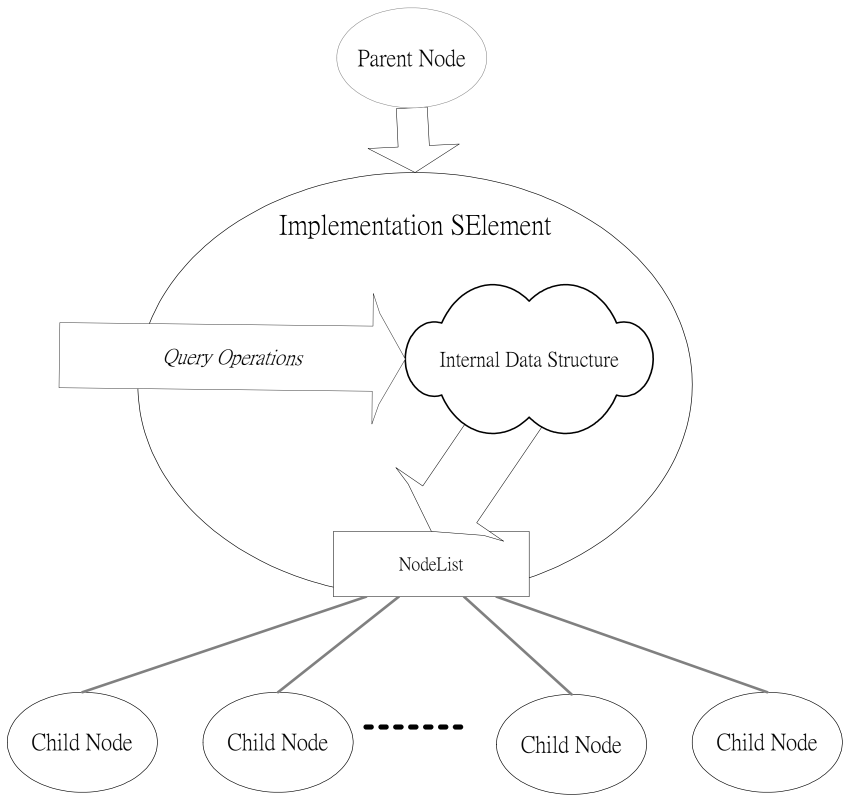


Figure 5.3 Structure of an Implementation SElement

Constructor method

The Implementation SElement objects are always created and wrapped inside Generic SElement objects. The constructor method takes a data element and a schema element as parameters.

Each class contains an internal data structure (e.g. an Rtree). The internal data structure will be initialized with the parameters specified in the schema element, or with the attribute values in the data element which links to the parameter by the schema. Section 4.2.1 describes the difference between a parameter type and an attribute type in the schema.

queryMethod()

The `queryMethod()` method publishes the implemented query operations for a SElement. The method does not take any parameters. It returns a `NodeList` of `Elements`, and each `Element` wraps details for one query operation. These Elements are named *query wrappers*, and will be re-used in submitting queries to the target SElement. Details of query wrappers will be covered in Section 5.4.1.

query()

The `query()` method accept queries from the user. Since the set of possible query operations for different structure types will be different and can not be known in advance, the `query()` method is a general query method which takes a *query wrapper* and the particular query operation to be invoke is specified in the query wrapper.

The query operation being called will get the required parameters from the wrapper. Then the query operation will be performed using the internal data structure. Three types of results can be generated from a query operation:

- Simple values (string, number, etc.);
- Composite values (an XML Element); and

- Child nodes of current SElement node.

In all cases, the results will be wrapped in a *result wrapper*, which will be covered in Section 5.4.1.

The first type of result is often generated when querying a property for the data structure as a whole, e.g. to query the number of nodes in a tree.

The second type of result is generated in more complex queries. For example, when a retrieving a tuple from a (relational) table alongside with a projection operation to select some of the wanted columns, a “new” structure will be generated as the result.

The third type of result means that the internal data structure eventually points to the child nodes of the current SElement, and a query may follow the data structure to get the child nodes as results. As multiple of child nodes could be returned in a query, the set of child nodes will be returned in a NodeList. Since the NodeList is only usable within the running application, additional function is implemented to convert the “pointers” of data Nodes into XPath representation.

Beside the listed three types of results, however, there is another additional result type of mixed “structured values” and “child nodes”. This is because a structured value may point to some childe nodes in some part of it, while the remaining part of structure is newly created and contains simple values. Nevertheless, there is no special at all, except that all references to child nodes are encoded as XPath values in `<node></node>` XML Element.

5.3 XML Parsing and Data Binding

Parsing is the process of converting the unstructured sequence of characters representing an XML document into the structured components of XML syntax: declarations, comments, elements, attributes, processing instructions, and characters. There are two basic models for actually processing the data in the XML source:

- An event-based parser generates a sequence of syntactic events, such as the "start of a `<document>` element", and feed the event into a consumer process. The set of "call-back methods" are defined under the Simple API for XML (SAX).
- A tree-based parser reads the nested element structure of the XML source and builds a tree-like data structure. The tree-like data structure can then be manipulated through the API associated with the tree data type. The structure components and the associated operations are defined under the W3C Document Object Model (DOM).

On the other hand, there is a rise on the use of data binding technology instead of ordinary XML parsers. Data binding technology automate the mapping between XML documents and Java objects by compiling an XML schema into one or more Java classes to handle the details for XML parsing and formatting. Sun Microsystem's Java Architecture for XML Binding [34] (JAXB) is a data binding API and it does receive a lot of attention in the Java-XML developers' community.

In our research, we enhance the W3C Document Object Model by combining its tree-like data structure with features in data binding technology. This enhancement maintains a good balance between the flexibility of DOM and the usability of data binding technology.

Similar to the data binding technology, our modeling also uses the XML Schema as hints to bind XML data into Java classes. However, instead of generating Java classes by compiling the schema, our Java classes use pre-defined models and pre-built Java classes, which offers much more complicated manipulations with supports in the internal data structure.

5.3.1. Parsing Process

The parsing process of SEOM Document involves both XML document and the Schema document. To begin with, both documents are parsed into DOM Document objects by DOM parser. The resulting Document objects are called *data element* and *schema element* respectively.

The SEOM Document constructor takes the data element and schema element as parameters. The data element will be assigned as the root node of the Document; the schema element will also be processed to retrieve the SElement declarations. For each SElement declaration, the `name` attribute represents the type name of XML Elements to be converted: a node of the schema element, which represents the current SElement declaration, and a node of data element, which matches the name specified in the declaration, will be passed to the constructor method of Generic SElement as a new data element and a new schema element respectively. Figure 5.4 illustrate the parsing process of SEOM Document.

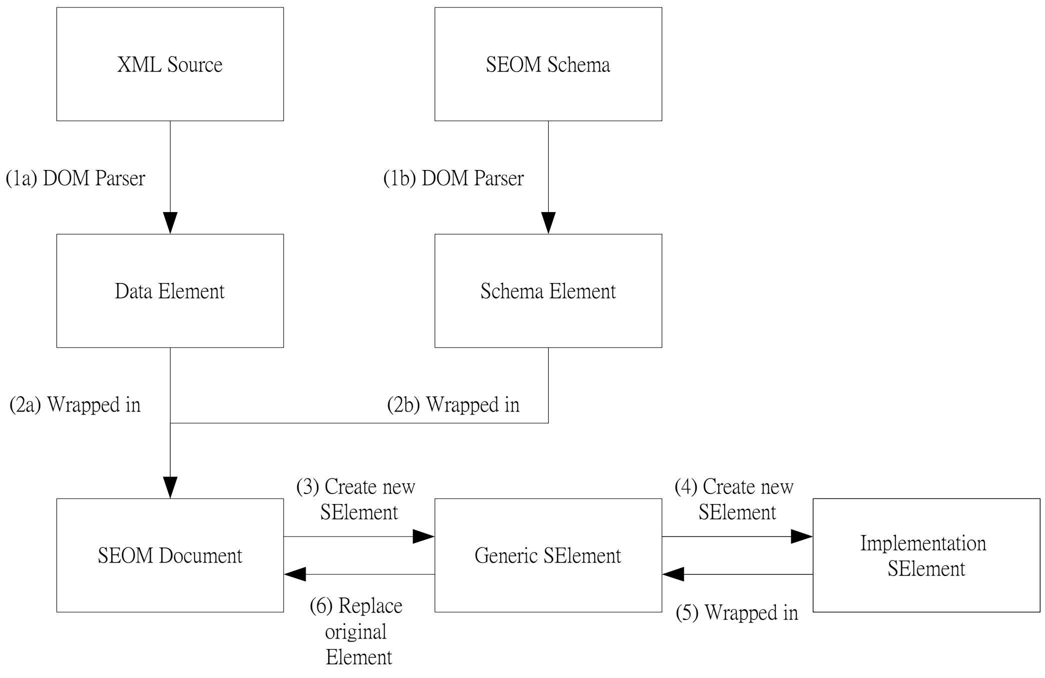


Figure 5.4 Parsing Process of SEOM Document

As mentioned in section 5.2.3, a Generic SElement wraps an Implementation SElement that actually implements the data structure. The class constructor of Generic SElement will fetch the needed Java class of the Implementation SElement according to the `class` attribute of the schema element, and instantiate the Implementation SElement object with the data element and schema element.

The constructor method of Implementation SElement takes a data element and a schema element as parameters. In the Implementation SElement constructor method, the data element will be validated against the schema element. Then the internal data structure will be initialized with the parameters specified in the schema element or in the data element, and the data values in the data element will also be copied to the internal data structure. However, only Elements declared as `RootNode` type and `InternalNode` type will be read and converted to the internal data structure; `LeafNode` type Element and its subtree will be attached to the SElement as its direct children and they will be reference by the internal data structure using their index number.

The returned Implementation SElement instance will then be wrapped as a member of the caller, i.e. the Generic SElement object. Moreover, the Generic SElement object will replace the original Element in its position, whereas the original Element will be detached and threw away. The SEOM Document is constructed by repeating the operations of replacing XML Element with SElement for all Elements that matches the declaration in the Schema.

5.4 Querying

There are two different approaches in querying an SEOM Document. The *query wrapper* approach is based on interactive exchanging of messages between the client user and an SEOM Document instance, while the SEOM-Extended XPath is developed based on the *query wrapper* by encoding the information as a string that can be embedded in the XPath.

5.4.1. Query Wrapper and Result Wrapper

A *query wrapper* is basically an XML Element that encapsulates the information of the query. A wrapper is represented by the XML Element `<query></query>`, and there are two required parameters, `path` and `queryMethod`. Therefore the skeleton of a query wrapper looks like:

```
<query path=”” queryMethod=””>
</query>
```

For the attribute `path`, it specifies the target of performing the query by XPath representation. The `path` is represented in the unique XPath expression from the document root which will return a node set of one node. The unique XPath expression uses the XPath index operator to restrict the path if multiple elements with the same name occur on the path.

For the attribute `queryMethod`, it is a keyword that represents the query method to be invoked. Different SElements may implements different query methods, and thus have different keywords. To retrieve a list of query methods available for current SElement, a query wrapper with null or empty `queryMethod` value can be submitted.

Get Query Methods

Since there could be more than one query method implemented in an SElement, and different SElements implement a different sets of query methods, it will be necessary to retrieve the list of query methods available on a particular SElement object before the user submitting a query.

To get the list of query methods, a query wrapper with null or empty value in `queryMethod` attribute is used. For example:

```
<query path=”/root/myRtree”>
</query>
```

or

```
<query path=”/root/myRtree” queryMethod=””>
</query>
```

could be used to retrieve the list of query methods available.

Consider that the returned result will usually used in submitting another query to the current SElement, it will be useful to make the result in the form of query wrapper such that the users can submit the subsequent queries more easily. Assuming that the SElement "/root/myRtree" has three query methods: `exact`, `range`, and `knn`, a request on the list of query methods on the SElement may results in a `NodeList` of three XML Elements:

```
<query path="/root/myRtree" queryMethod="exact"></query>
<query path="/root/myRtree" queryMethod="range"></query>
<query path="/root/myRtree" queryMethod="knn"></query>
```

However, besides the query method name itself, a query also takes a number of parameters. Therefore, the results must also include the parameters needed for a specific query method. As a result, we will put the parameters need for a query method as a child Element of the query wrapper of that query method, and a outer wrapper Element `<queries></queries>` will be added as the root, thus the previous list of query methods becomes:

```
<queries>
    <query path="/root/myRtree" queryMethod="exact">
        <x/>
        <y/>
    </query>
    <query path="/root/myRtree" queryMethod="range">
        <x1/>
        <x2/>
        <y1/>
        <y2/>
    </query>
    <query path="/root/myRtree" queryMethod="knn">
```

```
        <point/>
        <k/>
    </query>
</queries>
```

When the client receive the list, the XML source may be converted to a DOM structure, then the user can easily select a `<query></query>` subtree, fill the parameters, and submit another query to the SElement. In order to avoid being too complicated in usage, the parameter Elements accept any valid XML structure as values and will be validated at the server side, and the usage of each parameters have to be documented elsewhere.

Submit Query

After getting the NodeList holding the list of query wrappers, a particular query method can be selected by identifying its `queryMethod` attribute. For example, the first method of the previous example is selected, i.e.

```
<query path=”/root/myRtree” queryMethod=”exact”>
    <x/>
    <y/>
</query>
```

With a query wrapper, values of parameters can be inserted into the corresponding child Elements. For simplicity, the valid range/type of values for a parameter is manually documented elsewhere. Following is the query wrapper for the query method “exact” with *x* equals to 3 and *y* equals to 4:

```
<query path=”/root/myRtree” queryMethod=”exact”>
    <x>3</x>
    <y>4</y>
</query>
```

Query Results

With a query wrapper, users can submit queries to the SElement by filling the parameters in the wrapper Element (e.g. `<x></x>` and `<y></y>` in the example) and then invoking the `query()` method on the SElement with that query wrapper. The SElement will execute the query with the parameters, and three are three types of possible return values:

- Simple values (string, number, etc.);
- Composite values (an XML Element); and
- Child nodes of current SElement node.

In all cases, the results will be wrapped in a *result wrapper*.

For a result wrapper, it is basically an XML Element `<results></results>`. It wraps the results of query as well as the corresponding query wrapper such that a remote client can always identify the query-result pairs. Individual results are wrapped in XML Elements `<result></result>`. For the embedded query wrapper, its *path* attribute and `queryMethod` attribute is moved to the result wrapper's root. Following is the skeleton of a result wrapper for the query in previous section:

```
<results path="/root/myRtree" queryMethod="exact">
    <query>
        <x>3</x>
        <y>4</y>
    </query>
    <result>...</result>
    <result>...</result>
    ...
</results>
```

For the result values, they will be put in the <result></result> Element. Storing the first and the second result types are similar in a sense that they are ordinary XML data and can be naturally embedded in another XML Element. However, for the third type, an XPath representation is needed because the Child node itself cannot be embedded for two reasons:

- There is one and only one parent Node for any Element, therefore the Child node cannot have the original SElement and the `<result></result>` Element as parents at the same time.
- Duplication of a node on the tree may results in unpredictably large subtree, and thus a very long XML document.

This XPath representation use the unique XPath expression, moreover, it must address the need for the result type of mixed "structured values" and "child nodes". A solution to this is to use an additional Element `<node/>` with attribute `path` and no child Element. SEOM namespace is added to avoid name collision with Elements in the result data. Consequently, we add the namespace definition to the result wrapper and to the SEOM-specific Elements in the wrapper. For example, the previous example results in two child Nodes will looks like:

```
<seom:results path="/root/myRtree" queryMethod="exact"
    xmlns:seom="http://www.cse.cuhk.edu.hk/~ckma1/SEOM">

    <seom:query>
        <seom:x>3</seom:x>
        <seom:y>4</seom:y>
    </seom:query>

    <seom:result>
        <seom:node path="/root/myRtree/data[1]"/>
    </seom:result>
    <seom:result>
        <seom:node path="/root/myRtree/data[3]"/>
    </seom:result>
```

```
</results>
```

For queries returns other result types, the `<seom:node/>` Element will be replaced by the results. The `<seom:node/>` Element is also used to represent a node on the Document composite value types.

5.4.2. Embedding in XPath

The primary purpose of XPath is to address parts of an XML document. It uses a compact, non-XML syntax to facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax, and XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.

XPath also provides functions for manipulation of strings, numbers and Booleans. These functions provide us with more sophisticated, subtler ways of making selections within the source tree. The use of XPath functions takes us beyond being able to simply select nodes by names. Moreover, it is likely that the implementation of XPath function can be optimized and is more efficient that the location path.

Query Function

To make XPath works on the SEOM Documents, we extend the XPath functions to allow it making queries to the SElements. We will introduce a new XPath function `query()` to facilitate the need for making queries in SEOM Documents.

To submit a query, we have to specify a few parameters:

- a target SElement node,
- a method name, and
- a set of parameters specified in name-value pairs.

Therefore, a query() will has the form:

```
query("/document/SElement", "exact", "x=3", "y=4")
```

where the first parameter is an unique XPath expression pointing to an SElement node; the second parameter is the method name, which is the same as the value of attribute `queryMethod` in an query wrapper; the third parameter and all the following parameters are strings that representing the name-value parameters pairs for the `exact` query method and the name and value are separated by a "=" sign.

However, as stated in the XPath specification, an XPath expression can only results in one of the following four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

Since composite result is not possible to be represented, query functions which yields composite result is disabled.

Selecting Nodes with Predicate

In the XPath convention, selecting a subset of nodes from a node-set is often done by filtering the node-set with a predicate.

In an XPath expression, the location path consists of a sequence of one or more location steps separated by "/" and it selects a set of nodes relative to the context node. A location step consists of an axis, a node test, and zero or more predicates. The node-set selected by the location step is the node-set that results from generating an initial node-set from the axis and node-test, and then filtering that node-set by each of the predicates in turn.

Since the axis and node-test is defined on the tree structure itself, it will be more appropriate to add query functions by extending the predicate.

A predicate is specified in a pair of square brackets. It filters a node-set with respect to an axis to produce a new node-set. For each node in the node-set to be filtered, the predicate expression is evaluated with that node as the context node. The result of expression will be converted to a boolean. If the result is a number, the result will be converted to true if the number is equal to the context position and will be converted to false otherwise; if the result is not a number, then the result will be converted as if by a call to the boolean function. Thus a location path `para[3]` is equivalent to `para[position()=3]`.

In order to integrate the SElement query in the XPath expression, the query methods for SElements have to be embedded as function calls in a predicate expression with the left nodes of the SElement as the context node. Since the context node is given, the target SElement is known (which is the parent axis of the context node) and would not be necessary to include as a parameter in the query function. The following example represents a query on the SElement `myRtree` which have left nodes with node type `data`:

```
/myRtree/data[query("exact", "x=3", "y=4")]
```

This XPath will retrieve a set of Element with name `data` and they are the child nodes for the SElement `myRtree`. Now, the first parameter is the method name, and other parameters are the strings that represents the name-value parameters pairs for the `exact` query method.

As the predicate filter the nodes use a boolean value, the function call should results in a boolean value too. To achieve this, the context position is implicitly passed into the function. The function call then evaluates the query as a set of indexing numbers corresponding to the set of resulting nodes (as in the query wrapper approach). The context position is compared with the set, returns true if it is existed in the set, and false if it is not.

Chapter 6.
An Web-based SEOM Document Query System

6.1 Web-based SEOM Document Query System

In our work, we aimed at building a XML query system that based on the native XML approach. The SEOM Document Query System is a variation of ordinary XML query system in which the XML documents are transformed into the SEOM data model for querying. The SEOM extension in XML Schema will include metadata to specify XML elements in SEOM, which will be mapped to Java classes and being instantiated in the server space

Instead of mining the implicit structure information from documents [14], our approach in explicitly declaring the structural metadata allows classes for different structured-elements to be implemented, which may provide a different set of query operations. The structured data will be constructed as SElement, such that queries can be made to the internal data structure.

6.2 Client-Server Architecture

Our Web-based SEOM Document Query System adopts the client-server model. Sever and clients are separate nodes in a network, and the simplified interaction processes between them are as follows:

- Client sends a request for information or action to the server;
- Server receives the request and generates respond;
- Client receives the respond and display the information to the user.

For the communication between server and client, the HyperText Transfer Protocol (HTTP) is used. And the information being exchanged is encoded as XML messages.

Figure 6.1 shows the architecture of the Web-based SEOM Document Query System. The leaf hand side shows the server components, and the left is the client side. Simply speaking, the server is a Java application that contains the SEOM Document instances and provides allow the clients access the Documents via the Internet. The client is a web browser that allow the users input queries and view the results.

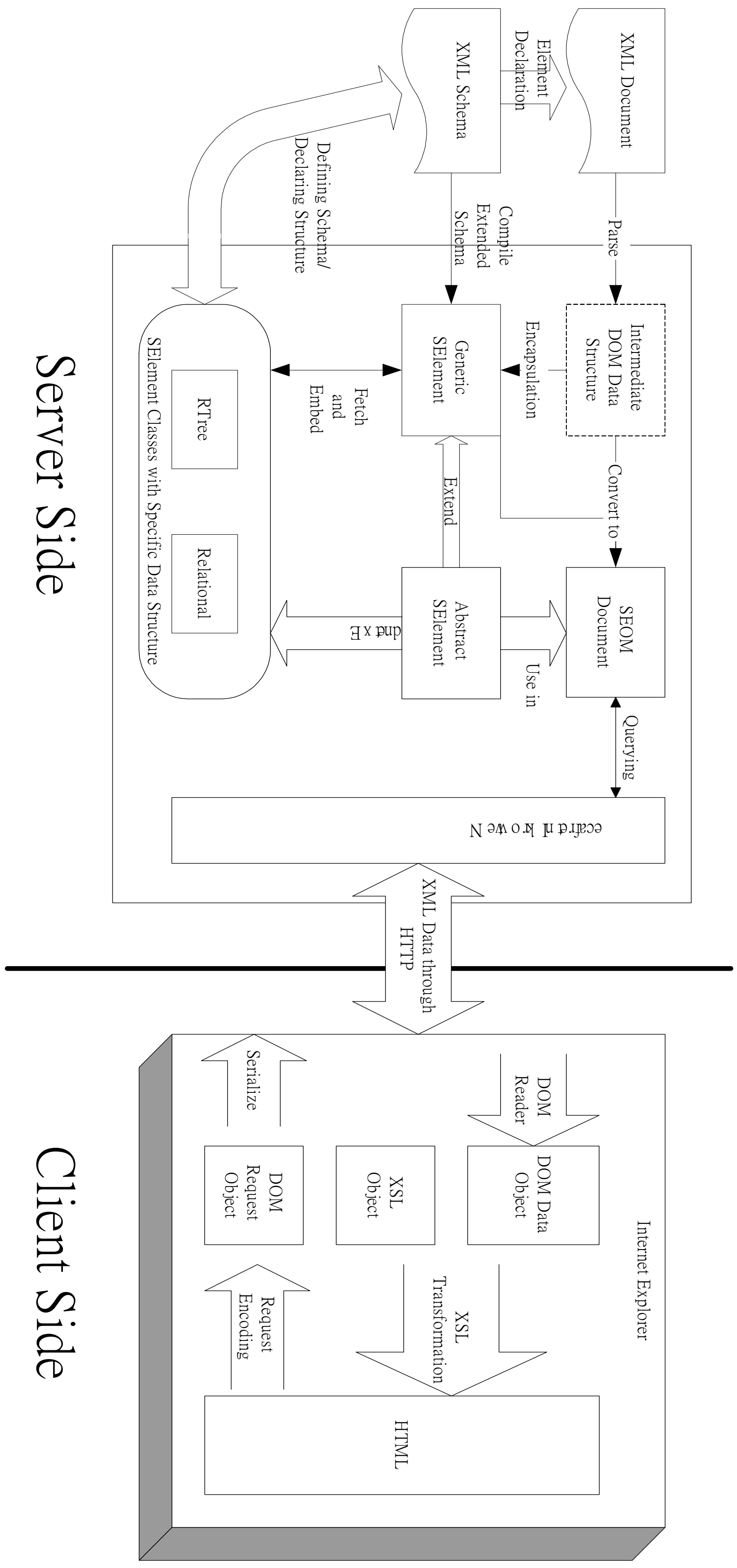


Figure 6.1 The Architecture of the Web-based SEOM Document Query System

6.3 The Server

The server program is a Java application developed on the J2SE platform. It handles the processing logics for the query system. Its major functions include data storage, and handling client requests. We will reveal the server processes in the following sub-sections.

6.3.1. Data Loading

The purpose of a query system is to query for information from a data repository. Our system is targeted in querying XML documents using the SEOM model. The documents are associated with SEOM Schema to define the data structure model of the elements.

On starting up the system, the server will load a set of XML documents from the document directory. A configuration file maintains the list of documents to be loaded and associate the documents with "document names", which may or may not be the same as the document's filename. The documents are parsed into DOM data structures, which will be converted into SEOM Document. The parsing process of SEOM Document is described in section 5.3. After loading the list of XML documents into SEOM Document objects, the system is ready for being queried.

6.3.2. Implemented SElement – R-Tree

In our system, an SElement which can represent an R-Tree [15] is implemented.

The R-tree is one of the most cited spatial data structures for indexing spatial data type. It is a modification of B-tree. It is a balanced tree that splits the space into rectangles which can overlap. Each node except the root contains from m to M children, where $2 \leq m \leq M/2$. The root contains at least 2 children unless it is a leaf. Figure 6.2 shows an example of r-tree of order 3.

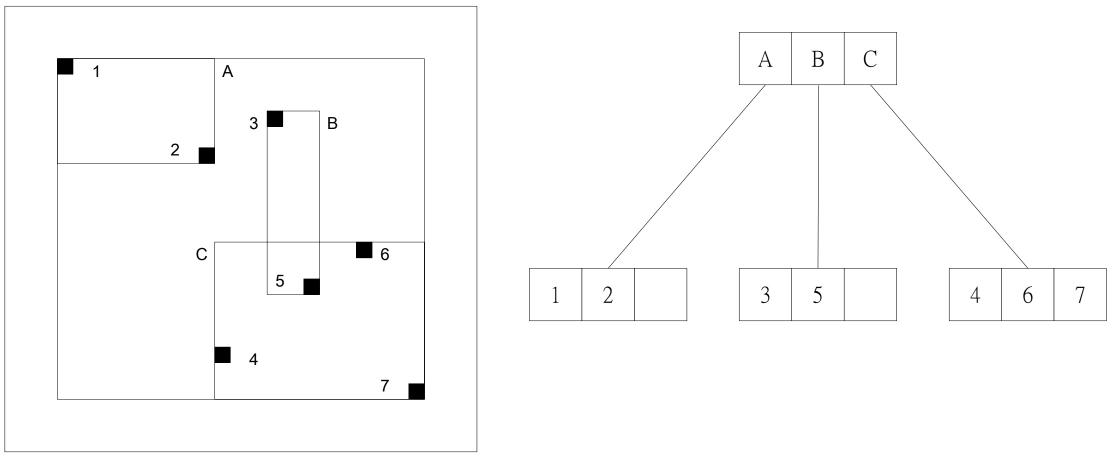


Figure 6.2 R-Tree

The node represents the minimum bounding rectangle containing all the objects of its subtree. Each of children of the node is split recursively. Pointers to the data objects are stored in the leaf nodes. As the bounding rectangle can be overlapped, it could be necessary to search more than one branch of the tree. Thus it is important to separate the rectangles as much as possible. The problem is handled by using some kind of heuristics during the insertion of data objects such as inserting the data into a leaf node which will cause minimum changes in the tree.

Query Operations

There are three main types of query operation on a R-tree:

- Exact Search
- Region Search
- K-Nearest Neighbors Search

For exact search, a coordinate tuple is given as parameter. All minimum bounding rectangles that cover the point is searched to see if a data point exist at the given coordinate. Assuming the element "`/rtree`" is an Rtree SElement, the query wrapper for exact search has the form:

```
<query path="/rtree" queryMethod="exact">
```

```
    <x/>
    <y/>
</query>
```

For region search, a rectangle is given as parameter by specifying the coordinates of its bounding faces. Minimum bounding rectangles that have overlapping area with the given rectangle is searched. Any data points existed in the given rectangle is returned. The query wrapper for region search has the form:

```
<query path=”/rtree” queryMethod=”region”>
    <x1/>
    <x2/>
    <y1/>
    <y2/>
</query>
```

where x1 < x2, y1 <y2 and they defines the box for the searching region.

For k-nearest neighbors search (knn-search), one of the data point is given as parameter, and a number k is specified too. The k nearest neighbors is found by using a branch-and-bound R-tree traversal algorithm:

- The nearest neighbor distances are initialized to ∞.
- A depth first traversal is begun at the root.

(i) at leaf level:

1. compute the distance between data is computed
2. if the distance is smaller then previous found distances, then update the distances to the new one

(ii) at non-leaf level:

1. generate the branch list

2. sorted the branch list based on the minimum distance to the input point

3. branches with distance larger than all the nearest neighbor distances will be dropped

4. iterate the remaining branches recursively, and go to (i) or (ii) accordingly.

The query wrapper for knn-search is:

```
<query path=”/rtree” queryMethod=”knn”>
    <point/>
    <k/>
</query>
```

where `<point></point>` will take the XPath for the given point, and the value for k is specified in `<k></k>`.

Schema

An example of schema for R-tree is given in below, explanations are given in comments (i.e. text between “`<!--`“ and “`-->`”)among the codes.

```
<seom:selement name=”myRtree” class=”Rtree”>
<!—the Rtree is binded to tag-name “myRtree” in XML document-->

    <se:rootNode>
        <se:attribute para=”BRANCH_FACTOR” value=”3”/>
        <se:attribute para=”LEAF_NUMBER” value=”3”/>
        <!— BRANCH_FACTOR and LEAF_NUMBER are internal variables
            of  the Rtree class which represents the maximum number
            of branch  and leaf in a node. They are assigned the
            value of 3 here -->
        <se:value>
            <se:internalNode ref=”rtreeInternal” maxOccurs=”3”/>
        </se:value>
    </se:rootNode>

    <se:internalNode id=”rtreeInternal” name=”node”>
        <se:attribute para=”NORTH” name=”N”/>
        <se:attribute para=”SOUTH” name=”S”/>
        <se:attribute para=”EAST” name=”E”/>
        <se:attribute para=”WEST” name=”W”/>
        <!--NORTH, SOUTH, EAST, WEST are internal variables of
            the Rtree class which represents the minimum bounding
            box. They are bind to attributes N, S, E, and W in the
            XML document -->
        <se:value>
            <xsd:choice>
                <se:internalNode
                    ref=”rtreeInternal” maxOccurs=”3”/>
                <se:leafNode ref=”rtreeLeaf” maxOccurs=”3”/>
                <!—The children of an internal is either internal
                    node or leaf node -->
            </xsd:choice>
        </se:value>
    </se:internalNode>

    <se:leafNode id=”rtreeLeaf” name=”data”>
        <se:attribute para=”X” name=”x”/>
        <se:attribuet para=”Y” name=”y”/>
        <!—X and Y are the coordinates of an data object, and
            they are bind to attributes x and y respectively -->
        <se:value>
            <xsd:any/>
```

```
            <!—The child of a leaf node can be any valid XML type
                as defined by <xsd:any/> -->
        </se:value>
    </se:leafNode>
</seom:selement>
```

Example SElement

Assuming the schema defined in previous section, we will give a piece of XML data corresponding to the definition as an example.

```
<myRtree>
    <node N="7" S="3" E="6" W="1">
        <node N="7" S="5" E="2" W="1">
            <data x="1" y="5">Tokyo</data>
            <data x="2" y="7">Beijing</data>
        </node>
        <node N="4" S="3" E="6" W="4">
            <data x="4" y="3">Hong Kong</data>
            <data x="6" y="4">Bangkok</data>
        </node>
    </node>
    <node N="6" S="4" E="16" W="8">
        <node N="6" S="5" E="16" W="9">
            <data x="9" y="5">Paris</data>
            <data x="16" y="6">London</data>
        </node>
        <node N="5" S="4" E="11" W="8">
            <data x="11" y="5">New York</data>
            <data x="8" y="4">San Fancisco</data>
        </node>
    </node>
```

```
</myRTree>
```

6.3.3. Network Interface

The network interface of the server is responsible to accept requests from the Internet, decode the message, pass the query to target document, then collect the results and reply to the client.

In the modular system design, the SEOM Document object is separated from the accessing interface. Whereas a GUI can be implemented directly over the SEOM Document to form a local query system, we implement the network interface such that the content of SEOM Documents could be accessed over the Internet. The implemented network interface is using the HTTP protocol with POST method.

During initialization, the network interface binds itself to a custom network port and listens to the port continuously. Whenever a request is received, a new thread will be created to handle the request. A Java InputStreamReader will be used to get the contents of the request. The HTTP header is not used, and the message body will be passed to a DOM builder to form a wrapper object. The wrapper object is the same as those describe in the previous section, except that a Document name will be added at the beginning of the `path` attribute to identify the instance of SEOM Document to be invoke. For example:

```
<query path="myDocument1/root/myRtree"/>
```

refers to the Element of XPath `/root/myRtree` in the document named `myDocument1`. The document name in the “path” attribute will be removed when network interface calls the target document's query method, i.e. when the wrapper becomes:

```
<query path="/root/myRtree"/>
```

The result returned by SEOM Document is a NodeList of Elements/SElements. The Network Interface will serialize the NodeList into XML-formatted text stream before sending back to the request client. There are two types of request, one is the query for available methods, another is to query for nodes using a given method. For the first one, the results returned by SEOM Document are a list of query method wrappers. An outer wrapper will be needed to make a well-form XML document, therefore we will encapsulate the list with a pair of `<reply></reply>` tags as in the following:

```
<reply>
    <query path="/root/myRtree"
        queryMethod="nearestNeighbour">
        <x/>
        <y/>
    </query>
    <query path="/root/myRtree" queryMethod="DOM"/>
    <query path="/root/myRtree" queryMethod="Data"/>
</reply>
```

For the later one, the results returned by SEOM Document are a NodeList of children of the target Element/SElement. The pointers to the children Nodes have to be represented in XPath such that the client can use the XPaths to reference to the Nodes. The XPaths are put in the `path` attribute of `<node/>` Elements. For example, in a query to `/root/myRtree` which results in the child Elements of idnex 1, 3, 4, and 7, the reply will be:

```
<reply>
    <node path="/root/myRtree/data[1]"/>
    <node path="/root/myRtree/data[3]"/>
    <node path="/root/myRtree/data[4]"/>
    <node path="/root/myRtree/data[7]"/>
</reply>
```

After sending out the reply, the processing thread will end and the network connection to the client will be closed. New connections will be established in subsequent requests.

6.4 Client Side

For our web-based query system, a web browser functions as the client platform and allows easy access to the server's information through the Internet. The client's major function is to interact with the user, such that the user can submit queries to the server and view the results. These functions are integrated in a HTML page by using scripting language and other XML technologies.

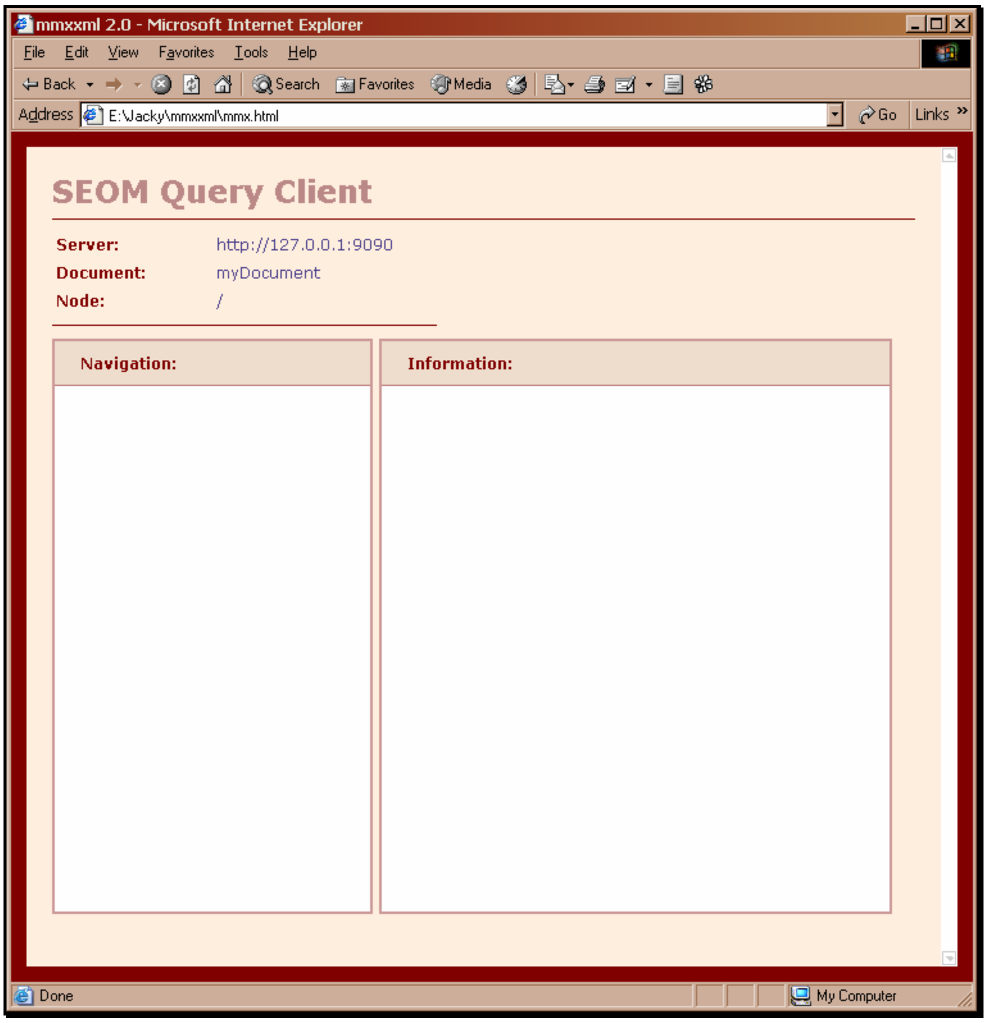
6.4.1. The Interface

We choose the Microsoft Internet Explorer 6.0 as the client platform. It has good support for XML. It has "XML data islands" to store XML content inside a HTML document, and "XML ActiveX objects" to manipulate XML data programmatically. These XML supporting feature makes it an ideal platform for the web-client platform of an XML-based systems.

A single HTML page is used to construct the interface of our web-client (Figure 6.3). It embeds the program codes to control the message flow and the XSL stylesheet to control the presentation of XML data.

On the user interface, user can specify the target server, the SEOM Document to be queried, and the target node (Figure 6.4).

There are also two panels, the Navigation Panel (Figure 6.5) and the Information Panel (Figure 6.6). When a node is visited, its available query methods will be retrieved and being displayed at the Navigation Panel. User can then select on of the query methods from the list. A dialog box will pop up to prompt for required parameters. Then the request will be sent to the server. The server will process and reply the request. When the reply contains a set of nodes, the nodes will be displayed at the Navigation Panel for the user to select; or when the reply contains other results or information, it will be displayed at the Information Panel.



mmxxml 2.0 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address E:\Jacky\mmxxml\mmx.html
Go Links
SEOM Query Client
Server: http://127.0.0.1:9090
Document: myDocument
Node: /
Navigation:
Information:
Done
My Computer

Figure 6.3 User Interface of the Web-Client

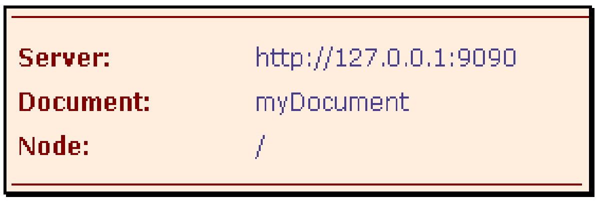


Figure 6.4 Server Address, Document Name, and Context Node

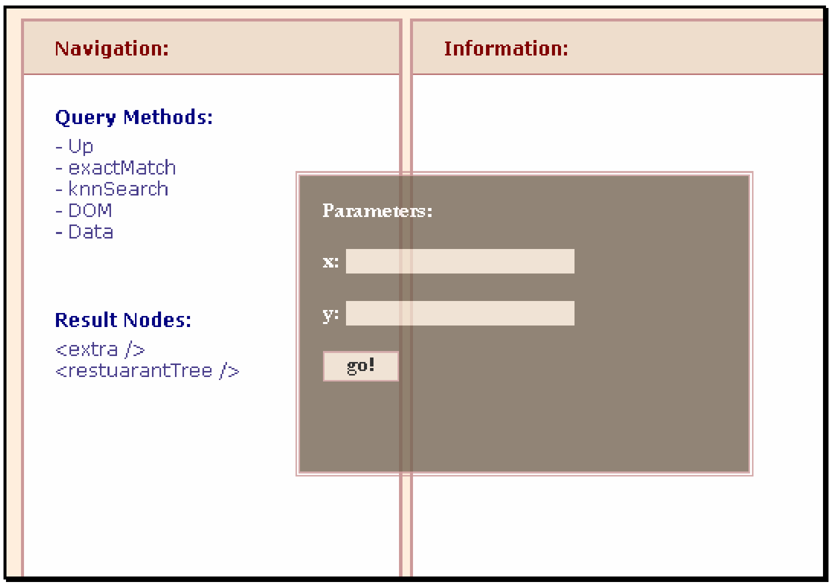
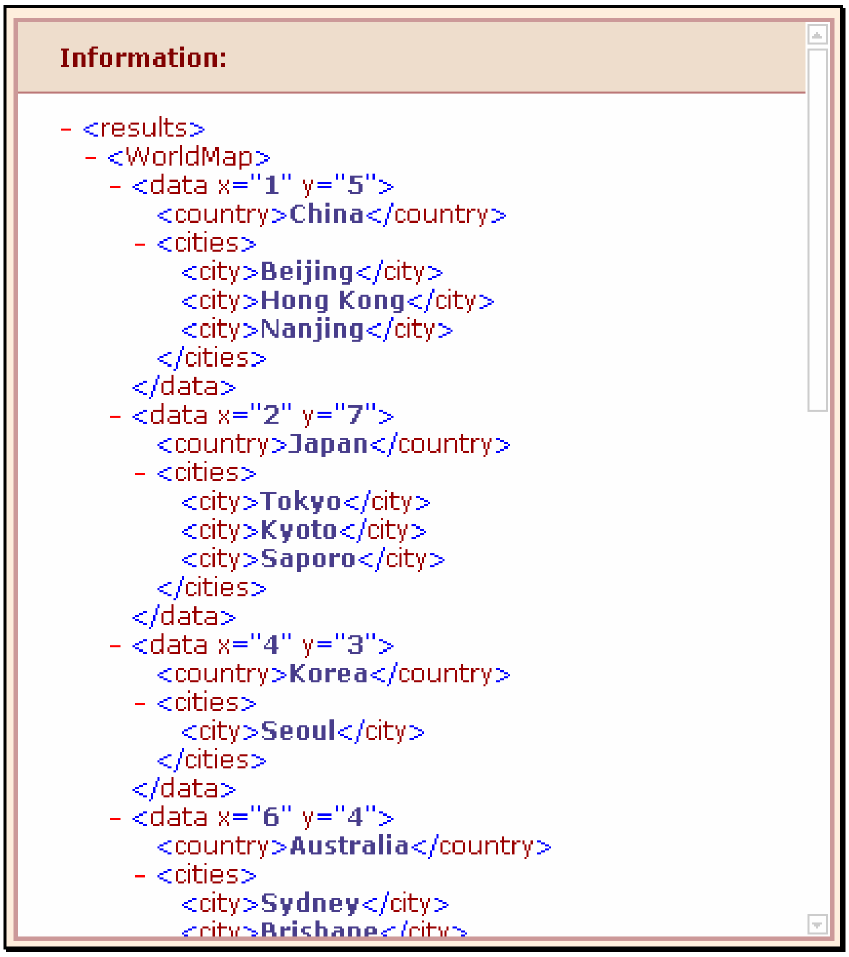


Figure 6.5 The Navigation Panel and the Parameter Dialog



```
- <results>
  - <WorldMap>
    - <data x="1" y="5">
        <country>China</country>
      - <cities>
          <city>Beijing</city>
          <city>Hong Kong</city>
          <city>Nanjing</city>
        </cities>
      </data>
    - <data x="2" y="7">
        <country>Japan</country>
      - <cities>
          <city>Tokyo</city>
          <city>Kyoto</city>
          <city>Saporo</city>
        </cities>
      </data>
    - <data x="4" y="3">
        <country>Korea</country>
      - <cities>
          <city>Seoul</city>
        </cities>
      </data>
    - <data x="6" y="4">
        <country>Australia</country>
      - <cities>
          <city>Sydney</city>
          <city>Brisbane</city>
```

Figure 6.6 The Information Panel

6.4.2. Programmatic Controls

Java Scripting technology is used in the HTML page to control the interaction between user and the server. Figure 6.7 shows the program flow of the client:

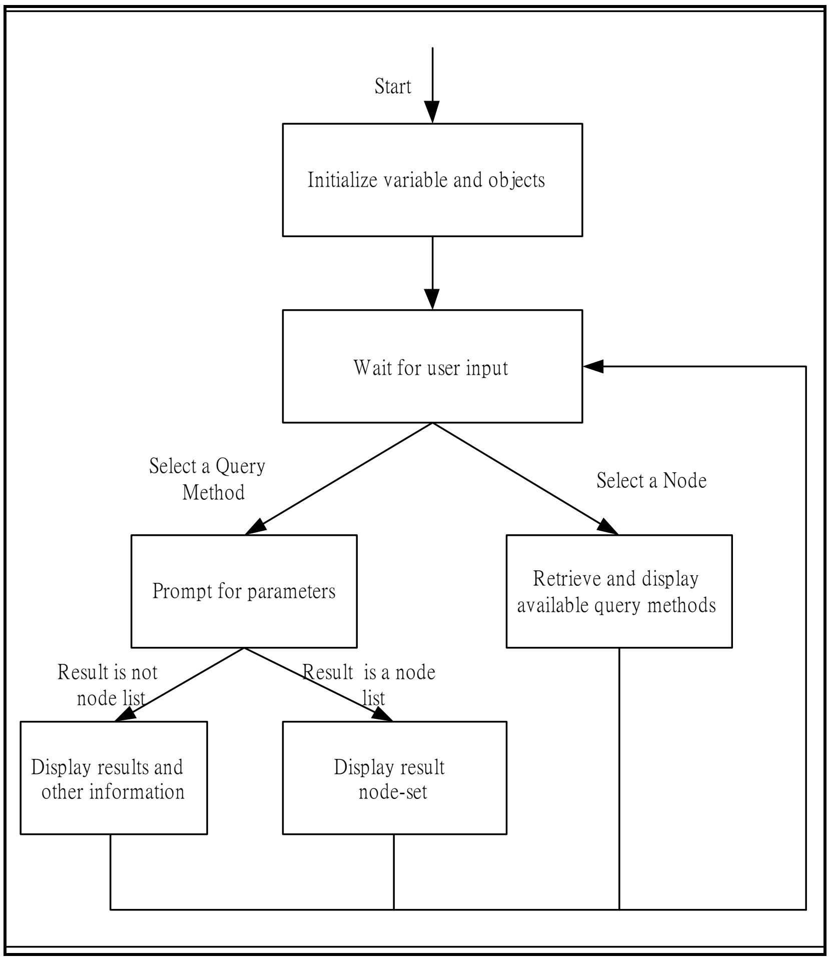


Figure 6.7 Program Flow of Client

When the client starts, the internal variables and ActiveX object will be initialized, and the interface will be draw. User may specify a server and a document to be queried, then the root of the document will be retrieved. User may then interact with the system by submitting a query or selecting a node (node selection is possible only after a query operation). The client will submit queries to the server, and update the available query methods, the node list, or the results in the Information Panel according to the server's replies.

XML Islands

With Internet Explorer 5.0 and higher, XML data can be embedded within HTML pages in Data Islands using the `<xml>` tag.

We have use the XML Islands to store the XSL stylesheets. By using the `transformNode()` method of XMLDOM ActiveX object, these stylesheets transform the XML data received from the server to a format that can be rendered by the browser. Figure 6.8 shows a simplified XSL stylesheet in XML Island which is responsible for displaying the parameter dialog box.

```
<xml id="dialogXSL">
   <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
   <xsl:template match = '/'>
      Parameters:
      <xsl:for-each select="*">
         <xsl:node-name/>:
         <xsl:element name="input">
            <xsl:attribute name="type">text</xsl:attribute>
            <xsl:attribute name="onchange">
               changePara("<xsl:node-name/>", this.value)
            </xsl:attribute>
         </xsl:element>
      </xsl:for-each>
      <a onClick="submit()">go!</a>
   </xsl:template>
   </xsl:stylesheet>
</xml>
```

Figure 6.8 XSL Stylesheet in XML Island

XML ActiveX Objects

The XMLDOM ActiveX Object and the XMLHTTP ActiveX Object is used at the client side to manipulate the XML data.

The XMLDOM ActiveX Object resembles the DOM interface. It exposes methods for loading and saving XML, accessing the document tree, creating new nodes, etc.

The XMLHTTP ActiveX Object, on the other hand, is responsible for accessing XML data from a Web server. It facilitates parsing the data into XMLDOM object, and post XML data directly to a Hypertext Transfer Protocol (HTTP) server.

When submitting a request, a message is first constructed by using the XMLDOM object. The XMLDOM object will then be serialized using the `XMLDOM.xml` property and sent out using the POST method provided by the XMLHTTP Object. After sending out the request, the browser will wait for the respond from the server. The respond is retrieve using the `XMLHTTP.reponseText` property. The XMLDOM's `loadXML()` method will then convert the respond message into a DOM structure.

These two XML-relate ActiveX Objects abstract the manipulation of XML data, which makes the web browser capable to act as the client platform for handling complicated XML data.

Chapter 7.

Evaluation

In this chapter, we will evaluate our research project in several perspectives. Since XPath is the official and the most popular accessing method for XML document, we will first present the experiments of querying spatial data in a XML document using XPath versus using an RTree model under SEOM. Then we will compare our model with several existing XML accessing methods features-by-features. Moreover, we will evaluate the advantages and disadvantages in applying our model, and raise some possible enhancements to strengthen its applicability.

7.1 Experiment with Synthetic Data

In order to demonstrate how a suitable data modeling speeds up query responds, we carried out experiments to compare making spatial queries by SEOM and by XPath. The data sets are objects in a two-dimension space. For each set of data, it will be prepared in two forms, one is optimized for XPath query, and another is optimized for R-Tree query under SEOM. For the later one, a packed R-Tree is built from the static data set which reduces the overlapping between the MBRs of the R-Tree.

To measure the performances, we test the document construction time and the random seeking time for both models:

- Document construction time - measure the time taken to construct the representation of a document.
- Random seeking time - measure the time of random seeking.

The test framework uses the approach of running the document construction test some number of times (10 for the results shown) on a document, tracking the best and average times for that test. When the document construction test is completed, we move on to the random seeking time test on the same document. After both tests have been completed on one document, the process repeats with the next document. There are 10 documents of each model with increasing document sizes, starting from 100 data objects, and doubling the data size in each next document, up to 51200.

Figure 7.1 shows that the Document Construction Time of both XPath and SEOM are growing roughly linear to the growth of data size, while the SEOM impose a larger overhead. This is concise to our expectation, since both XPath and SEOM are based on a DOM structure.

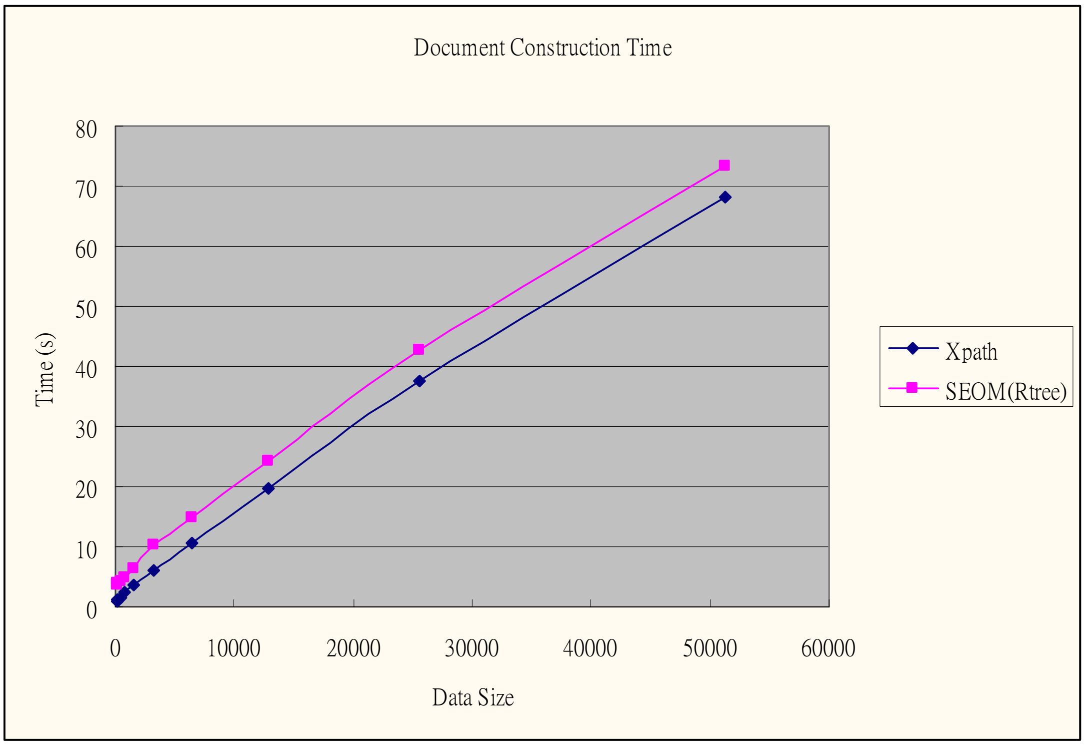


Figure 7.1 Document Construction Time

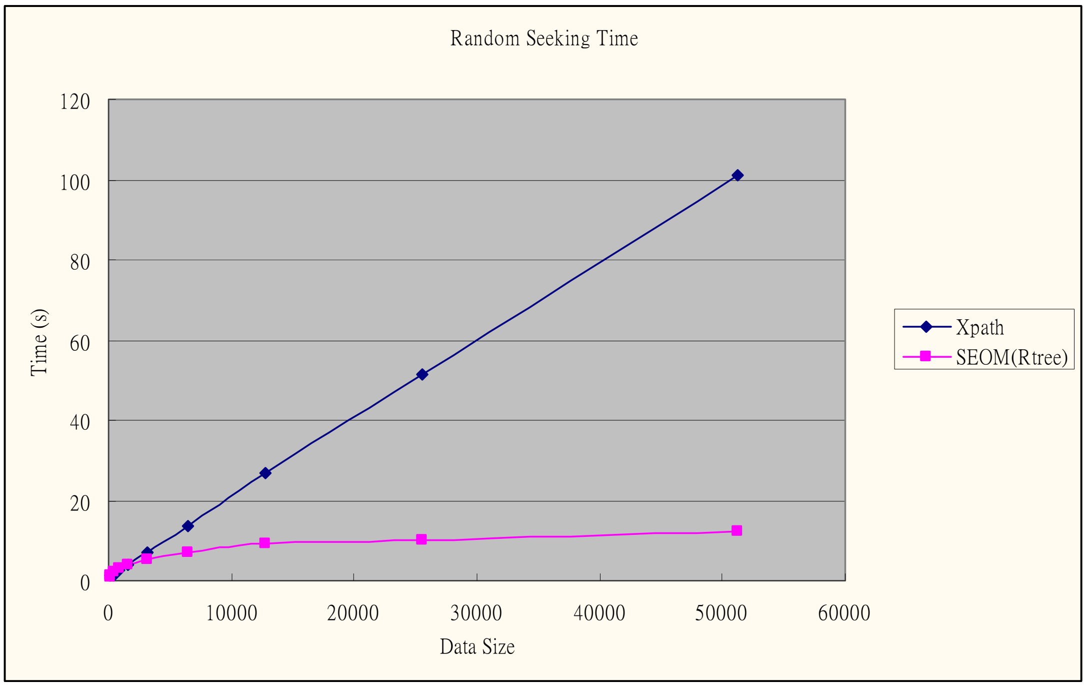


Figure 7.2 Random Seeking Time

Figure 7.2 shows the relationship between the Random Seeking Time of XPath and SEOM using the Rtree model. The graph shows that the XPath seeking time increases linearly with data size, while the SEOM achieves a much better performance than linear growth. The result is expected since SEOM uses an indexed structure (i.e. RTree) instead of sequentially searching the records as in XPath.

The experiments show that SEOM out-performs conventional accessing methods for XML files. However, each Data Model aims at processing a specific type of data and is not applicable to generic XML data, and the XML data have to be pre-processed into the way as specified by the corresponding Schema, which could be a time consuming process too.

7.2 Qualitative Comparison

Besides the improved searching performance, SEOM also provides additional features that were not provided in current XML accessing techniques. Table 7.1 summarizes some features in SEOM and two other popular XML query languages:

	SEOM	**XPath**	**XQuery**
Schema awareness	Yes	No	Yes
Functions support	Yes	Limited	Yes
XML-based syntax	Yes	Yes	No
Indexing Structure	Yes	No	No
Accelerated Query Processing	Yes	No	Partial
Expressive Power	Medium	Weak	Strong
Jointly Uses with non-XML data sources	Yes	No	No
Applicability	Specific uses	Generic uses	Generic uses
Data Model	Tailored complex nodes (SElement)	Nodes & Branches only	Nodes & Branches only

Table 7.1 Comparison of XML accessing methods

7.3 Advantages

Our Structured-Element Object Model for XML data described in this thesis has certain strengths and weaknesses. We will discuss its pros and cons in this and the coming section.

Generally speaking, our model enjoys the following advantages:

- It separates logical data representation from the physical data representation. The modeling of multiple XML elements as a single logical unit hides unnecessary details that programmers would not be interested in. This object encapsulation approach coincides with the object-oriented programming paradigm, and favors modular development and reuse of software components.
- It defines a set of schema constructs for mapping multiple XML elements into a single SElement object. Besides providing metadata in parsing process, the schema allows different applications to process the same piece of XML data in

the same way and facilitate information exchange.

- It separates data representation from objects implementation; the object can be optimized the process a specific type of data. Moreover, data can be validate, and the internal dependence can be checked when an SElement object instantiates.
- It is based on the DOM model and still shares the DOM interface. Many technologies that are applicable to DOM can be used in SEOM also. XPath and XSLT are examples of these technologies.
- The schema is loosely tied to the SElement. This allows greater flexibility in the SElement implementation, and thus results in better extensibility.

These advantages are very important in XML application development. Our model combines the features from Document Object Model as well as data binding technologies, which must be a powerful tool for the programmers to development better XML applications.

7.4 Disadvantages

Though our model has many advantages for representing complex data objects in XML, it also has some drawbacks, however. Here listed below are the disadvantages of our model.

- The overhead in parsing XML file into SEOM Document is longer than parsing the file into DOM Document. This is because the SElement has to check the validity of the XML data, and it has to build and indexing structure instead of simply encapsulates the data value in an Element.
- Our model relies on using a schema, which may brings sort of inconvenience. Moreover, our schema is not an official one like the W3C DOM, fortunately, the use of XML namespace makes it possible to have both type of schema exists together, and this is our approach.

Actually, these disadvantages are the tradeoffs of some good features. Though building of index structures add extra workload to the system, they provide great improvement in efficiency in use. Also, the use of schema can easily to be employed in other XML applications, hence increasing the extensibility and flexibility of the model.

7.5 Means of Enhancement

Despite that our model enhances the DOM to make it more usable with complex data object, there are rooms for improvement. Some of the possible enhancements are listed below:

- Currently the model is addressing data access only. It is possible to include other manipulation methods in the SElement model. Operations for inserting child nodes, removing child nodes, may involve changes in the internal data structure; the SElement model will have to define the constraints for these operations.
- Currently the schema only defines a hierarchical relationship for the nodes. It is possible to include referencing mechanisms (e.g. XPointers) to form a graph relation. Graph representation is more general and expressive that the tree representation.

Chapter 8.
Conclusion

In this thesis, we have presented the Structured-Element Object Model for XML data. The model combines features from the Document Object Model and the data binding technologies. It adopts the tree structure of DOM; meanwhile, it introduces the additional SElement type, which represents logical entities as the objects used in data binding technologies. The SElement resembles the interface of a DOM Element, and it implements additional interface to allow queries being made to its internal data members. The SEOM Document brings great flexibility in representing complex data objects under a hierarchical organization.

In order to bind the physical XML data representation to the Java classes that implement SElement type, we define a schema for declaring SElement objects. The schema not only specifies the element structure of an XML element that corresponds to the root node of an SElement object, it also specifies how the XML elements (of different tag names) will be mapped to the target object's data member. In fact, the schema is not only used in parsing XML files. It provides a framework that people can interchange their information on structures. By compromising on a standard modeling and an interface specification, different application can process an XML document in exactly the same way, especially for application at two ends of a client-server system.

After presenting the classes that made up SElement type and the parsing process from XML file to SEOM document, we introduce a mechanism to support querying

SElements. The mechanism is based on exchanging of XML wrapper messages. We wrap the queries with required parameters in XML messages. The result values/structures/nodes are also wrapped in XML messages. This wrapper method is particularly suitable for interactive knowledge discovering in XML repository.

We also extend the XPath to support filtering of nodes using SElement. We introduce a query function for SElement nodes, which can be embedded into the predicate term of the XPath location step. The query function takes parameters and returns the resulting nodes as if a query is made to the SElement. The extended XPath is very useful and handy when the document structure is known.

Finally, we implement a web-based XML query system; the system uses SEOM at server side. We have implemented an R-Tree SElement and used it in an SEOM Document. The web browser client allows users to examine the SEOM document; we can choose different query methods at SElement nodes, submit queries, and select a node to navigate from the result set. This XML query system demonstrates the usability of our work.

To conclude, our contributions are:

- An object model combining the feature of DOM and data binding technology;
- A schema for mapping physical XML data into Java classes that implement logical entities;
- A mechanism to support querying SElements by exchanging XML wrapper messages;
- An extension of the XPath to support filtering nodes using the query function in SElement;
- A web-based XML query system using SEOM has been implemented to demonstrate our work.

Our modeling is generic and can be used for a wide range of complex data objects. It provides a concise and neat interface for accessing complex data objects and maintains the generic tree structure of XML document. With consideration to the parsing overhead, our mechanism is still very suitable to be applied to various XML applications, which will be the next generation data management technology.

Bibliography

1. Fabio Arciniegas A., *XML developer's guide*, McGraw-Hill, ISBN 0072126469, New York, 2001.

2. Kal Ahmed et al., *Professional XML meta data*, Wrox Press, ISBN 1861004516, Birmingham, UK, 2001.

3. Mohammad Akif et al., *Java XML programmer's reference*, Wrox Press, ISBN 1861005202, Birmingham, UK, 2001.

4. Suad Alagic, Institutions: Integrating Objects, XML and Databases, *Information & Software Technology*, Volume 44, Number 4, March 2002, pp. 207-216.

5. Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura, A Data Model for Temporal XML Documents. *Proceedings of 11th International Workshop on Database and Expert Systems Applications (DEXA'00), 6-8 September 2000, Greenwich, London, UK*, pp. 334-344.

6. Richard Anderson et al., *Professional XML*, Wrox Press, ISBN 1861003110, Birmingham, UK, 2000.

7. Dirk Bartels, A Comparison between Relational and Object Oriented Database systems for Object Oriented Application Development, FastObject White Paper, available at http://www.fastobjects.de/pdf/FO_WHIT_OOvsREL_30MAY01_EN.pdf

8. Elisa Bertino and Elena Ferrari, XML and Data Integration, *IEEE Computer Society*, November/December 2001 (Vol.5, No.6), pp.75-76.

9. Kurt Cagle et al., *Professional XML schemas*, Wrox Press, ISBN 1861003110, Birmingham, UK, 2001.

10. Tuong Dao, An Indexing Model for Structured Documents to Support Queries on Content, Structure, and Attributes, *Proceedings of the IEEE Forum on Reasearch and Technology Advances in Digital Libraries (IEEE ADL '98)*, April 22-24, 1998, Santa Barbara, California, USA, 1998, pp. 88-97.

11. Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, 27(3): pp. 59-74, 1998.

12. Joseph Fong, Francis Pang, and Chris Bloor, Converting Relational Database into XML Document, *Proceedings of 12th International Workshop on Database and Expert Systems Applications (DEXA 2001)*, 3-7 September 2001, Munich, pp. 61-65.

13. R. Goldman, J. McHugh, J. Widom, From Semistructured Data to XML: Migrating the Lore Data Model and Query Language, *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB'99)*, Philadelphia, Pennsylvania, June, 1999.

14. Mark Graves, *Designing XML Databases*, Prentice Hall, ISBN 0130889016, NJ.

15. A. Guttman, R-trees: A Dynamic Index Structure for Spatial Searching, SIGMOD'84, *Proceedings of Annual Meeting*, Boston, Massachusetts, June 18-21, 1984. ACM Press 1984, pp. 47-57.

16. David C. Hay, A Comparison of Data Modeling Techniques, *The Database Newsletter*, Volume 23, Number 3, May/June, 1995 (Updated 1999).

17. Jingyu Hou, Yanchun Zhang and Yahiko Kambayashi, Object-Oriented Representation for XML Data, *Proceedings of the Third International Symposium on Cooperative Database Systems and Applications (CODAS 2001)*, Beijing, China, April 23-24, 2001, pp. 43-52.

18. Zaijun Hu, Gerhard Vollmar: Towards XML Metamodel Patterns for XML Data Modeling. *Proceedings of 12th International Workshop on Database and Expert Systems Applications (DEXA 2001)*, 3-7 September 2001, Munich, Germany. *IEEE Computer Society 2001*, pp. 71-74.

19. Petr Kuba, Data Structures for Spatial Data Mining. *FI MU Report Series*, Faculty of Informatics, Masaryk University, September 2001.

20. H. Lin, T. Risch, T. Katchaounov, Object-Oriented Mediator Queries to XML Data, *Proceedings of the 1st International Conference on Web Information Systems Engeering (WISE'00)*, Vol.2, Hong Kong, China, 19-21 June, 2000, pp. 38-45.

21. The Lore Project. http://www-db.stanford.edu/lore.

22. F. Manola, Towards a Web Object Model, *OBJS Technical Report*, Object Services and Consulting, Inc. (OBJS), February 1998, http://www.objs.com/OSA/wom.html.

23. Hafedh Mili, Francois Pachet, Ilham Benyahia, Fred Eddy. Metamodeling in OO. *Workshop on Metamodeling in OO (OOPSLA'95),* Workshop Summary, pp. 105-110.

24. J. P. Morganthal, *Enterprise application integration with XML and Java*, Prentice Hall PTR, ISBN 0130851353, Upper Saddle River, NJ, 2001.

25. Mark Nadelson and Marina Evenstein, Undaunted Testing - Develop a testing architecture that boldly attacks complex nested code, *Java Pro*, http://www.fawcette.com/javapro/2002_08/magazine/features/mnadelson/, 2002.

26. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, Object Exchange Across Heterogenous Information sources, *Proceedings of the Eleventh International*

Conference on Data Engineering (ICDE 1995), Taipei, Taiwan, March 1995, pp. 251-260.

27. Ketan C. Patel. Storing XML Content. *XML magazine*. http://www.fawcette.com/xmlmag/2001_12/magazine/columns/collaboration/kpatel/.

28. Giuseppe Psaila, ERX: A Conceptual Model for XML Documents, *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC)*, Italy, March 19-21, 2000, Volume 2, pp. 898-903.

29. Auguste Rabarijaona, Rose Dieng, Olivier Corby, and Rajae Ouaddari, Building and Searching an XML-Based Corporate Memory, *IEEE Intelligent Systems*, Volume 15, Number 1, January/February 2000, pp. 56-63.

30. Andreas Renner, XML Data and Object Databases: A Perfect Couple? *Proceedings of the 17th International Conference on Data Engineering (ICDE 2001)*, April 2-6, 2001, Heidelberg, Germany, pp. 143-148.

31. Chantal Reynaud, Jean-Pierre Sirot, and Dan Vodislav, Semantic Integration of XML Heterogeneous Data Sources, *Proceedings of International Database Engineering and Applications Symposium (IDEAS 2001)*, Grenoble, France, July 16 - 18, 2001, pp. 199-208.

32. D. Riehle and T. Gross. Role model based framework design and integration. *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98)*, Vancouver, Canada, October 18-22, 1998. pp. 117-133.

33. Leonard J. Seligman and Arnon Rosenthal. XML's Impact on Databases and Data Sharing. *IEEE Computer*, Vol.34, No.6, pp.59-67, 2001.

34. Sun Microsystem, http://java.sun.com/xml/jaxb/index.html. *Java Architecture for XML Binding (JAXB)*.

35. Sun Microsystem, http://java.sun.com/xml/jaxp/index.html. *Java API for XML Processing (JAXP)*.

36. B. Surjanto, N. Ritter, and H. Loeser, XML Content Management Based on Object-Relational Database Technology, *Proceedings of the First International Conference on Web Information Systems Engineering*, Volume 1, 2000, pp 70-90.

37. Szentivanyi G., The Role of XML in Generic And Distributed Multimedia Management, *Proceedings of IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '99)*, 1999, pp 317-318.

38. Dan Wahlin. Back to Basics: The XML Fundamentals. http://www.fawcette.com/xmlmag/2001_11/online/online_eprod/XML/

39. Andrew Watt, *XPath essentials*, John Wiley & Sons, Inc., ISBN 0471205486, New York, 2002.

40. Jennifer Widom, Data Management for XML Research Directions, *IEEE Data Engineering Bulletin, Special Issue on XML*, 22(3): pp. 44-52, September 1999.

41. World Wide Web Consortium, http://www.w3.org/DOM/. *Document Object Model (DOM)*.

42. World Wide Web Consortium, http://www.w3.org/TR/xpath. *The Recommendation for XPath*.

43. World Wide Web Consortium, http://www.w3.org/XML/. *Extensible Markup Language (XML)*.

44. World Wide Web Consortium, http://www.w3.org/XML/Schema/. *XML Schema*.

45. Xyleme project. http://www.xyleme.com.