Compression of XML Data

This Report is submitted to the University of Strathclyde in partial fulfilment of the Regulations for the Degree of MSc in Information Technology Systems

Candidate
Mathias Neumüller
Reg. No. 200053493

Supervised by
John N. Wilson
Department of Computer Science
University of Strathclyde

September 2001

Except where otherwise expressly indicated the work reported in this document is my own. It has been performed during the month of June to August 2001 and has not been submitted for assessment in connection with any other award whatsoever.
ABSTRACT

Most implementation of the Document Object Model (DOM), which are used to represent XML data in the computer’s memory, suffer from extensive memory requirements. Thus large volumes of data are hard to handle. This work tries to analyse the reasons for this and designs a system that is specifically tailored for the need of large document. A DOM compliant data structure called DDOM, that uses dictionary compression to reduce the amount of memory being used, was designed, implemented and tested. Comparison with other implementations show a saving in the order of 30% to over 80% in terms of memory usage. The influence of different types of data on the compression ratio is shown. This report also identifies the the conceptual differences between relational and irregular data sources with respect to the possibilities of a compact representation.
# CONTENTS

1. **Introduction: XML and compression** .......................................................... 1

2. **The extensible markup language (XML)** ...................................................... 3
   2.1 Document structure, well-formed and valid documents ......................... 3
   2.2 Entities and their textual representation .............................................. 4
   2.3 Document object model (DOM) representation ....................................... 7
   2.4 The simple API for XML (SAX) representation ..................................... 8
   2.5 Other representations ............................................................................ 9

3. **Compressed forms of XML** ........................................................................ 11
   3.1 Minimising storage space requirements ................................................ 11
   3.2 Minimising transmission bandwidth requirements ................................... 12
   3.3 Minimising main memory requirements for processing ........................... 12
   3.4 Review: Related work ............................................................................ 13

4. **Concept and design** ................................................................................... 17
   4.1 Problem description ............................................................................... 17
   4.2 Assumptions ......................................................................................... 17
   4.3 Compression algorithm ......................................................................... 18
   4.4 Internal data structure .......................................................................... 19
      4.4.1 XML specific considerations ........................................................... 19
      4.4.2 DOM specific considerations .......................................................... 20
      4.4.3 Java specific considerations ............................................................ 21
      4.4.4 Dictionaries .................................................................................... 21
   4.5 Possible approaches ............................................................................... 22
      4.5.1 Object tree structure with uncompressed data .................................. 22
      4.5.2 Object tree structure with dictionary compression ............................ 23
      4.5.3 Table oriented structure with dictionary compression ..................... 23
      4.5.4 Linear structure with dictionary compression .................................. 25
## Contents

### 5. Implementation
- 5.1 Naming conventions, architecture and class hierarchy 
- 5.2 Dictionaries and structure array
- 5.3 Improving performance for query relevant methods
- 5.4 The modes of the nodes
- 5.5 Parsing and JAXP integration

### 6. Testing
- 6.1 Test data
- 6.2 Testing the internal structure of the implementation
- 6.3 Testing the logical structure of the DOM tree
- 6.4 Other tests

### 7. Example data
- 7.1 Domain name server database
- 7.2 Shakespearean plays

### 8. Measurements
- 8.1 Measurement procedure
- 8.2 Domain name database
  - 8.2.1 The influence of the document size
  - 8.2.2 The influence of the document style
  - 8.2.3 Comparison with relational data structures
- 8.3 Shakespearean plays

### 9. Conclusions
- 9.1 Achievements
- 9.2 Suggestions for future work
  - 9.2.1 Improvements on compression ratio
  - 9.2.2 Improvements on dictionary and parser implementation
  - 9.2.3 Extending the functionality

### References

### Appendix
- A. Debugging output from DDocumentFragment.dump()
- B. API Documentation
- C. Source code
1. INTRODUCTION: XML AND COMPRESSION

XML, the eXtensible Markup Language, is a current key topic in nearly all aspects of information technology. It is a way of describing semi-structured data. Beside the data itself, its semantics are encoded as well. The idea behind the development of XML through the world wide web consortium (W3C) was to create a universal way of exchanging data across system and vendor boundaries.

So far this design goal seems to be achievable as more and more applications are developed using XML as format for data exchange. Although the development is still in its early stages, a massive demand becomes obvious. But every thing comes at a price. The price to pay for versatility and exchangeability is the verbosity of XML. Data represented in XML format usually requires more space than represented in another, purpose-designed, typically binary-coded format. Since XML is primarily designed to exchange data over networks, i.e. the Internet, this represents a waste of resources, namely transmission bandwidth. It also creates some problems when working with the data locally, in case the document becomes too big to fit into the computer’s memory.
To overcome these problems, there is a large amount of work going into compressed forms of XML at the moment. On a second look it will become apparent, that one needs to distinguish between different contexts. XML is a general purpose term for a number of related standards and representations. The next chapter will introduce some important terms and representations of XML, before chapter 3 explains the ideas behind compressing it and what work has already been done in the field.
2. THE EXTENSIBLE MARKUP LANGUAGE (XML)

This chapter will give a brief introduction into XML and some related standards. This can not be complete, as XML comprises a extensive set of standards, but should suffice for the purpose of this project. Some of the concepts mentioned will be Java specific. Although it may seem odd to restrict something to a specific programming language at this stage, Java represents the first choice when implementing open standards as it is self developed as one. Therefore the use of Java throughout the project was a premise.

2.1 Document structure, well-formed and valid documents

Listing 2.1 shows an small example document conforming to the XML standard. It will be used to explain some properties and related terminology concerning XML. In general, every XML document will consist of a prolog, a document element and some optional white space, comments or processing instructions at the end. In case of the example, the prolog consists of the first three lines and everything below, enclosed by the <STUDENT> and
2. The extensible markup language (XML)

<xml version="1.0"?>
<!DOCTYPE student SYSTEM "student.dtd">
<!−− this file contains information about a student −−>
<STUDENT REGNO="200053493">
  <NAME>
    <SURNAME>Neumüller</SURNAME>
    <FIRSTNAME>Mathias</FIRSTNAME>
  </NAME>
  <COURSE>Information Technology Systems</COURSE>
  <DEGREE>Master of Science</DEGREE>
</STUDENT>

Listing 2.1: The student example XML document

</STUDENT> tags form the document element. Well-formed documents obey this structure with all of their individual elements satisfying the constraints as defined in the XML recommendation [22]. In the remaining part of the report the term “XML document” will be used in this sense. In addition, valid XML documents satisfy some further constraints. These additional constraints are defined by a Document Type Definition (DTD) or an XML schema. Although validation through a DTD can be helpful in order to achieve higher compression ratios, only well-formed XML documents will be assumed for the project.

2.2 Entities and their textual representation

The document in listing 2.1 shows several of the possible types of content an XML document can have:

XML declaration: The first line is used to mark the following as XML and must appear in all well-formed XML documents. Currently there exists
only the version 1.0 of the XML definition, though it comes in two distinct editions. Optional parameter, attributes, are allowable in the <<?xml> tag and are frequently used to define the character encoding used.

**Document type declarations:** The <!DOCTYPE> tag is used to declare where the DTD used to validate this XML document can be found. It is also possible to have the DTD included in the document itself.

**Comments:** Used to add some information for human readers to the document. The third line of the document represents a comment and is usually ignored by XML applications. Note that this comment is part of the prolog, but it could also appear as part of the document element or in the closing section of the document. It should serve the goal of readability only. Comments are enclosed by the tags <!-- and -->. 

**Elements:** These are used to form the structure of the document. The example contains several elements, such as STUDENT, NAME, etc. These must not be confused with their associated tags, which are used to indicate the beginning and end of an element, such as <STUDENT> and </STUDENT>. The element itself consists of these tags together with all enclosed attributes and any enclosed content. The element STUDENT in the example file fulfills a special purpose. It is the document element, the one and only element that is not enclosed by any other element.

**Attributes:** Elements can have associated attributes, like the attribute REGNO
of the **STUDENT** element. They are enclosed in the opening tags of the associated elements and and their values need to be enclosed in quotes. Although attributes are defined in a more complicated way, they will be treated as a key-value pair for the remainder of this project report, with additional notes where necessary.

**Text:** All data in an XML document must be stored in textual form. The literal string “Mathias” is an example of an text element. Obviously not all characters can appear in an ordinary text element, as some characters carry a special meaning like the `<` and `>` sign. These can either be enclosed in specially marked section, CDATA sections, which will not be parsed, or be represented using entity references.

**Entities and entity references:** Entities are used to represent predefined content. This can also be used to represent characters that can not be entered in a normal way, because the character in question is not available on the keyboard or is forbidden because it has a special meaning, like the angle brackets. There are different kinds of entities: internal, external and predefined entities. Entities need to be defined before they can be used, usually in a DTD. They can be referenced to by using entity references. An example of an entity reference is the umlaut ’¨’ referred to as ’&uuml;’ in the content of the **SURNAME** element. The actual entity declaration is contained in the DTD, but is out with the scope of this report.
Fig. 2.1: The sample document represented as a DOM tree

Other things may appear within a XML document. There are processing instructions, Notations and CDATA sections. They are hardly used in real-world XML documents and would not serve the purpose of clarity at this stage. Formal definition for these and the other entities can be found in [22].

2.3 Document object model (DOM) representation

The document object model (DOM) represents a logical view of an XML document. Most XML elements introduced above are represented by an object. These objects have certain properties and methods. The definition of the DOM [20] declares these objects, their properties and behaviors. It does not imply a specific implementation though. However, there are some name-bindings included for Java, to avoid multiple implementation with different
names for identical functionality. These are given in the form of interfaces, i.e. method signatures an implementing object has to use.

Of course the content of an XML document remains unchanged, even if it is represented in a different way. The DOM structure makes the hierarchical structure of XML obvious. Any document has exactly one element at its highest level, the document element. This element can have an arbitrary number of children, which in turn can have children again. That means that any XML document can be represented as a tree.

Figure 2.1 shows the example document as fully expanded tree. There are some noticeable changes between the document shown in listing 2.1 and its DOM representation caused by the parsing process. The entity reference ‘&uuml;’ was resolved by the parser and is now represented by the character ‘ü’. The comment and other meta-information in the prolog have disappeared. The hierarchical order of the document becomes visible. Note that although the attribute REGNO is shown on the same level as the children of the STUDENT element in the used JTree representation, it is none. According to the DOM, attributes are properties of their owning elements, but not their children.

2.4 The simple API for XML (SAX) representation

This representation follows a completely different route to represent the content of an XML document. It was not designed by the W3C, but by members of the XML-DEV mailing list. The concept of the DOM representation requires the entire XML document to be parsed into a DOM tree, before it can
be worked with. This can sometimes be a waste of resources, if one is only interested in a small subset of the document. It was also designed without a specific implementation or programming language in mind, although object orientated features were used. Therefore adaptation of the DOM structure to a specific programming language may not utilise its specific strengths. Several programmers working with XML addressed these shortcomings and came up with their own approach to XML and Java. Rather than generating an internal representation from the source document, they produced a series of events, a concept typical for Java. The events generated by the SAX parser, like the beginning or end of an element, can be handled by other parts of the program. This lightweight approach results in better performance but is limited in its capabilities. Although it is supported by many software developers it is not an official standard.

Although this fundamentally different approach is not useful in terms of the objectives of this project, a SAX parser will be used in the implementation to create the compressed structure. It is therefore worth noting the general concept at this stage.

2.5 Other representations

There are some other ways of representing XML documents, which will be of little to no importance to this project. One emerging standard is the JDOM API [9], developed by Jason Hunter and Brett McLaughlin. It defines an API that is more consistent and logical from a Java programmers point of view. It also tries to integrate the different APIs that already exist. The
Java community process has defined JAXP [12], the Java APIs for XML Parsing. It integrates SAX and DOM implementations into a framework that allows parser independent development and defines standardised methods for parsing. However, it does not define its own representation of XML documents.
3. COMPRESSED FORMS OF XML

Since the suitability of a compression method will depend on the actual representation and the intended use of the data, it will be necessary to distinguish different scenarios here. The first section aims at storage, the second at transmission of and the third at work with compressed forms of XML. Each of these is associated with its specific requirements. Finally there is a brief review of projects already existing in these areas.

3.1 Minimising storage space requirements

The requirements for storage are simple: Compress XML text files without loss of information to the biggest extend possible.

Information expressed in form of textual XML will take considerably more space than the same information stored in a purpose designed format. This is due to two factors: The limitation to a textual representation increases the storage requirements of non textual data. The string of characters representing the number “127” takes 3 bytes in most encoding schemes, while the value could easily be represented using only one byte using conventional integer encoding. The second factor is the mark-up itself. This kind of meta-information is usually suppressed in other file formats as it is known (usually by definition) what kind of content appears at what position in the docu-
ment. This is what makes XML more flexible, but also makes its files bigger. Considering cases where the structure is relatively simple, for example data from a relational database, the repetition of the same structure tags for each row of data is a considerable waste, especially if the structure is already known in advance, for example by definition in the DTD.

### 3.2 Minimising transmission bandwidth requirements

The requirements for transmission of XML are similar to those of storage with some extensions. First of all the time needed for compression and decompression is more crucial than in the first case, as real-time processing is likely to be needed. This also implies the need for on-the-fly compression. An algorithm that has to read the entire document before it can be compressed will generally be of limited use for a communication application. One also needs to worry about standardisation and availability of resources across system boundaries. If a DTD is used to improve the compression ratio as done by XMill [11] described later, this DTD must be accessible at both ends of the communication link. If it is not available locally at the receiving end, it needs to be transferred before the transmission of the compressed data can begin.

### 3.3 Minimising main memory requirements for processing

Inside the computer’s main memory the requirements become different. Although it is possible just to transfer the textual representation from secondary memory to primary memory, possibly using some of the compression methods
described below, this is hardly a suitable form for processing. Computers do not understand text in the same way humans do. XML needs to be parsed before it can be queried, modified or analysed. Hence the usual representation in memory is the DOM. The different content elements are represented as different objects. These objects store the data of the elements they represent together with a suitable set of methods which can be applied on it. The document is not any longer a stream of characters but a tree of specific objects. In this model, individual nodes must be accessible. This would generally be impossible, if one would compress the document as a whole.

3.4 Review: Related work

There are many projects aimed at the compression of textual XML data. A first choice is to use an all purpose compression/decompression program such as GZIP [5]. Most compressor will achieve quite significant compression ratios on XML data. The algorithms used internally are well known and based directly upon Shannon’s Information Theory [14]. However, since they only compress a stream of tokens, the compression ratio is not optimal. Exploiting properties specific to XML, the compression ratio can be improved even further.

XMill [11] splits structure and content of an XML document, thus allowing better compression rates using conventional compression algorithms on both separate parts. Millau [6] does the same but also encodes tag and attribute names using binary token, as suggested for the Wireless Application Protocol (WAP) [18]. Depending on the structure of the document, slightly
different algorithms are used. If a DTD is available, the compression of the structure can be improved by just noting the distance from a standard form.

All these methods, although far developed, have a significant disadvantage: They only compress whole documents. In order to work with the document, it must be decompressed first. Compressed XML of this kind can not be queried without prior decompression.

As explained above the DOM is more suitable to represent XML data in the computer’s memory. There exist several implementations of DOM compliant parsers, such as the two Apache projects Xerces [17] and Crimson [16]. They and many other implementations have in common that the parsed version of the XML, the DOM tree, consumes ever more memory than the (uncompressed) textual XML document as will be shown later in chapter 8. This is due to the fact, that all values are still stored in textual form and that the API infers a relatively complex interface. As the hierarchy does not follow from the position in memory any more, a series of links between the different nodes is required to maintain the logical structure of the document.

If the size of the parsed document exceeds the available memory, the parser usually aborts with an exception. Thus documents from a certain size upwards can not be handled by these implementations. Nevertheless, the data does not have to be in the computers main memory to satisfy the recommendations of the W3C DOM working group. The internal representation of the data is completely up to the developer of the specific DOM implementation. To address these problems, Gerald Huck developed the Persistent DOM (PDOM [8]). This DOM compliant data structure keeps most of the data in a file on the computers mass storage system. Only the part that
3. Compressed forms of XML

is currently worked with is held in a buffer memory. Access to parts out with the buffer is enabled by automatic swapping processes. Data requested is loaded into the buffer, while other parts that are not longer in use are written back to the file. Since hard disks usually used as second memory of a computer exist in nearly arbitrary sizes, practically all documents can be handled. The price for this is the relatively slow access time to individual nodes. If the node is not in the buffer, relatively slow access to secondary memory is required. PDOM supports compression of entire documents held on disk, but no compression of individual nodes held in the buffer.

In the case of XML data generated from the content of a relational database, there is another possible way of allowing access to the data without exceeding the computer’s main memory. Using a DOM compliant interface layer capable of translating the DOM method calls into database queries, the data could remain in a conventional, usually disk-based database system whilst allowing access using the interface of the DOM. Again the translation and access to secondary memory is likely to consume considerably more time than access to a document kept in the computer’s main memory. But in case of a database system as described in the next section compact storage could be paired with flexibility of access.

Cockshot, McGregor and Wilson [3] designed a database system that compresses relational data using dictionary compression for every entry domain. The regular structure of relational databases allows to store the lexemes or keys for the dictionary entries for each column in a minimal sequence of bits, without losing the possibility to access individual fields. By doing so, the size of the data can be reduced by approximately one order, allowing more
data to be fitted into the computer's memory. Unfortunately, this only holds true for regular data and cannot be applied directly to semi-formal XML.

But the problem can be inverted. Confronted with the problems arising when querying XML data sources, Shanmugasundaram, Tufte, He, Zhang, DeWitt, and Naughton [13] developed an algorithm to transform XML to relational data and query it using conventional database engines. Although conversion is possible for practically all data sources and query types, the developed system was limited in performance as typically numerous joins are required to perform a specific query.
4. CONCEPT AND DESIGN

This chapter explains some of the considerations done at the design stage of the project. It concludes with two different design suggestions, from which only one was fully implemented.

4.1 Problem description

The purpose of this project is to compress the internal representation of XML data, the DOM tree, rather than the external textual XML file. Unlike most other developments reviewed in section 3.4, this project is aimed at a compressed form of the DOM to allow compact representation of parsed XML data in the computer’s main memory. This is a requirement to enable easy querying of the data. Queries should work on the compressed data itself rather than by decompressing each element in turn in order to access it. The emphasis lies more on easy and fast accessibility than on high compression rates.

4.2 Assumptions

There are some assumptions on which this project is based. The most important assumption is about the kind of data that is to be processed. It will be
assumed as “nearly” regular, i.e. regular with few exceptions. Although the
design should be capable of handling any well-formed XML document, com-
pression can only be expected for regular structures, containing redundant
data. Such XML documents are often referred to as “data-centric”. This
assumption seems to be valid for large documents, as such documents are
usually extracted from some kind of relational database. The assumption of
regularity will become invalid for small documents, but these are obviously
of minor interest in terms of compression.

The second assumption made concerns the proposed use of the data. It
will be assumed, that the data is mainly queried but not altered significantly
throughout its life-cycle. Although the design should not rule out any kind
of modification, the emphasis lies on retrieval.

4.3 Compression algorithm

The option to access individual elements of the DOM tree in its compressed
form in effect rules out any form of entropy encoding as it results in variable
length code, which makes access to individual elements impossible. A simpler
compression mechanism is the dictionary based compression. Rather than
storing the data in the different nodes of the document itself, only references
to a dictionary, the lexemes, are stored. The dictionary is commonly used
throughout a domain, thus avoiding repetitions of data. This is especially
important in a Java program, as String objects are extremely expensive in
terms of storage requirements.
4. Concept and design

4.4 Internal data structure

It is necessary to distinguish between the internal, physical data structure and the external, logical one. While the later is given by the DOM recommendation [20], the data structure used internally will determine the memory requirements.

4.4.1 XML specific considerations

The key to finding a memory-efficient representation of any data lies in a compact but unambiguous representation of the data. XML has two main properties which are responsible for the extensive memory usage requirements. Firstly its verbosity that binds the representation of data to a textual form. The second is redundancy. Tag names are repeated very frequently, even if the DTD or the XML standard itself does not permit any variation at a specific point. A closing tag for example must always have the same name as the last opening tag, otherwise the document would not be a well-formed XML document. Nevertheless, the tags name is repeated in the closing tag. This can be addressed by the dictionary based compression. By replacing each value with a short lexeme, the space required to store multiple occurrences of the same value is reduced. Of course this will also effect multiple occurrence of the same piece of information contained in the data.

The actual structure of XML documents allows constructs far beyond the possibilities of relational data models. In a relational database, the structure of the entries is known in advance and follows a simple two-dimensional scheme. Not so in XML. If you allow any well-formed XML, you will have to
cater for some extreme but perfectly standard-consistent cases. In XML it is possible for an element to have an element with the same name as one of its own children. Generally elements with similar names may appear in different contexts. A real-world example of such a document structure is shown in listing 7.3, where TITLE elements appear at two different contexts. The DTD shown in listing 7.2 actually defines the document structure in such a way. It would therefore be wrong to assume, that all elements with a certain name appear on the same level of the hierarchy or have children or parents of the same type. Although certain limitations apply, in general any XML element may have any other well-formed XML content as children. Since the children that are appended to an element are not known in advance, an element would always have to cater for the biggest possible child. This makes bit optimised representation as used in [3] impractical.

4.4.2 DOM specific considerations

Because it was the intention to use the DOM as mean of accessing the data, it is important to allow easy mapping between the DOM interface as defined by the W3C and the internal, compressed representation. It must be possible to get parent, child and sibling entries with acceptable computational expense. As the objects declared by the DOM form a special form of a tree, a tree-like structure appears to be desirable. Nevertheless it interferes with some of the other requirements, which will become apparent in the following sections.
4. Concept and design

4.4.3 Java specific considerations

Two things are well known to be responsible for poor memory efficiency of Java applications. Strings are internally stored using 16 bit unicode representation. Considering that most textual information only uses the 26 letters of the Latin alphabet, usually in a mixture of upper and lower case letters, and only few additional symbols, this seems to be a waste of memory. Nevertheless, unicode is a fundamental step on the way to platform independence and part of the DOM recommendation [20] and can not be ignored easily. Another performance disadvantage of strings in Java leads to a more general problem of efficient encoding. Strings are objects. With any kind of object there is a certain amount of overhead involved. References to objects need to be maintained somewhere in order to allow the garbage collector to do its job properly. Their internal state needs to be preserved. All that takes memory.

It will therefore be of importance to minimise the number of objects used to represent the XML data. This holds especially true for the number of strings needed.

4.4.4 Dictionaries

The previous design considerations have concluded that a dictionary compression based method seems to be desirable. It will be necessary to make some decisions about this vital part of the design. There are two distinct approaches. The first one is straightforward. Every string of text appearing in the document, regardless if it is an element name, the value of an at-
tribute or some text element is stored in one big dictionary. In this way, no
two identical strings need to be stored twice. The main disadvantage of this
approach is that searching and ordering of such a large dictionaries will be
slow. It also appears inappropriate to store element names, part of the docu-
ment structure, together with its containing data, possibly from a specialised
domain such as numbers. Although for the moment every piece of data is
treated as a string, this may change in the near future, as the XML Schema
definition [23] is about to introduce typed data. In this context it seems to
be desirable to store data from different domains in different dictionaries.

For these reasons the decision was made to use distinct dictionaries for
distinct domains. For the design chosen, a domain is specified by the type
of the node and may also have subdomains. This is especially important for
the dominant text elements. The content of such entries will be stored in
subdomains which are given by the names of their containing elements.

4.5 Possible approaches

The following paragraphs give a brief overview on how one could implement
a DOM compliant data structure. Starting from the most direct possible
translation of the DOM specification into a design it gradually improves the
design towards the goal of compact representation.

4.5.1 Object tree structure with uncompressed data

This section describes the most obvious, direct translation of the DOM in-
terface into Java. Though it does not compress the data in any form it is
mentioned for comparison and understanding. Every node is represented by an object. It stores all the information required to answer any of the possible method calls to it. A text node for example would need to store a reference to its parent node, its owning document and the text it is representing. This is easy to implement but expensive in terms of memory.

4.5.2 **Object tree structure with dictionary compression**

This approach is very similar to the one above with the exception that the textual data is stored in a centrally maintained dictionary. This avoids much of the memory being wasted compared to the previous approach but still leaves a large number of objects in memory (one per node plus some additional ones for the dictionary).

4.5.3 **Table oriented structure with dictionary compression**

This approach tries to minimise the number of objects held in memory. Although it is necessary to have one object for every node during its creation or when it is retrieved from a DOM structure, there is no need for these objects to persist for the entire life span of the document. As long as the data contained by them is maintained somewhere, they can easily be created on demand. As DOM trees are usually created by appending nodes sequentially to a document, only two objects need to exist at most times: The one that is appended and the one it is appended to (plus the document object as it is in fact the source or factory of any other newly created objects).
To exploit this feature, data of similar content is stored together in a table. For example there is one table for elements with a certain tag name. Because all the elements in this table have the same tag name, it only needs to be stored once. Only things that are different are stored per entry, like a list of children for each individual element. The objects representing the different nodes contain only a reference to the table and the entry within it. Once an element is appended to another one, the data is maintained by the corresponding table. Thus the object can cease to exist. If the parent element is asked for this node again, it will create a new object, referring to the same content. Because the data is held by central dictionaries and same tables comprising the structure of it but not by the newly created object, it will not use a lot of memory and can be disposed of after use without fear of data loss. Figure 4.1 shows the underlying table structure of a simple database containing two domain name server entries, a simplified example of the data sets described in section 7.1. Note that for a similar database with higher cardinality the number of tables used would not increase, but only the number of entries in some of the tables.

Although this design seemed to be worth implementing, early tests indicated significant technical problems with this solution. In addition the memory savings achieved by the use of dynamic object creation together with tables used to store similar content was small compared to the savings achieved by the use of dictionaries. Hence a simplified version was designed, reducing the amount of technical problems while maintaining a compact form.
4. Concept and design

Fig. 4.1: Design of the table oriented structure with dictionary compression for a database containing domain name server entries. Note: some of the references to the owning entries are not shown for clarity.

4.5.4 Linear structure with dictionary compression

The idea remains the same, only that all kind of nodes are now stored together in one table, an array. The textual data is still based by a centrally managed dictionary. But this time all the structure information is contained in a single object as well, the document object. Once again individual node objects hold only a reference to this structure and a index within it. They are created on demand and cease existing when no external reference to them is kept. In effect this translates the tree structure into a linear structure. Additional entries will be needed to maintain the hierarchy. This in fact is
4. Concept and design

Exactly the way the external representation, the XML file, looks like. Content is surrounded by start and end tags. In the end an XML file is a linear structure, a long series of characters. Thus the position of entries itself carries a meaning. The parent element does not need to be stored. It is found by going back to the previous node that is able to hold children. Of course the performance of such requests will suffer, as often (short) linear search processes are needed to perform a certain function. But on the other hand, a linear search in memory may still be faster than the access to a well defined place in external storage. Figure 4.2 shows the design using the student example. Note that the type of the entries in the structural table is stored in form of a short key. What exact type to use for this and the associated index entry will be discussed in section 5.2. It is also worth noting that the entry keys used in the dictionary are merely shown for clarification but not actually stored as they are implied by their position.
### Type and Design

<table>
<thead>
<tr>
<th>Type</th>
<th>Index</th>
<th># ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>START-DOCUMENT</td>
<td>-</td>
<td>1 STUDENT</td>
</tr>
<tr>
<td>COMMENT(^1)</td>
<td>1</td>
<td>2 NAME</td>
</tr>
<tr>
<td>START-ELEMENT</td>
<td>1</td>
<td>3 SURNAME</td>
</tr>
<tr>
<td>START-ATTRIBUTE</td>
<td>1</td>
<td>4 FIRSTNAME</td>
</tr>
<tr>
<td>TEXT</td>
<td>1</td>
<td>5 COURSE</td>
</tr>
<tr>
<td>END-ATTRIBUTE</td>
<td>1</td>
<td>6 DEGREE</td>
</tr>
<tr>
<td>START-ELEMENT</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>START-ELEMENT</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>TEXT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>END-ELEMENT</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>START-ELEMENT</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>TEXT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>END-ELEMENT</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>END-ELEMENT</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>START-ELEMENT</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>TEXT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>END-ELEMENT</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>START-ELEMENT</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>TEXT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>END-ELEMENT</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>END-ELEMENT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>END-DOCUMENT</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

### Figures

| # TEXT:STUDENT          | 1 200053493 |                          |
| # TEXT:SURNAME          | 1 Neumüller |
| # TEXT:FIRSTNAME        | 1 Mathias   |
| # TEXT:COURSE           | 1 Information Technology Systems |
| # TEXT:DEGREE           | 1 Master of Science |

**Fig. 4.2:** The structure of the example document stored in an array (left) together with the dictionaries containing its data (right). Note: \(^1\)The COMMENT entry in this structure is shown for completeness although comments are ignored by most parsers and would therefore not occur in practice. The associated dictionary is not shown in order to save space.
This chapter reviews some of the decisions that were made during the implementation stage. Even reduced to a functional subset of the DOM level 1 recommendation [20], the number of objects and methods to implement is quite big. For this reason it will not be possible to discuss all the code here. Instead only small pieces of it will be discussed in some greater detail.

5.1 Naming conventions, architecture and class hierarchy

Because the main feature of the DOM implementation to develop is its internal use of dictionary compression, the whole implementation will be called “Dictionary based Document Object Model” (DDOM). The objects implementing a specific interface of the W3C recommendation, will be named by the interface name, together with the preceding letter ‘D’, i.e. the interface Text will be implemented by a class called DText.

Figure 5.1 shows the architecture of the system developed. The core is formed by an implementation of the DOM. Note the special function of the DDocument class. Apart from being the root node of every DDOM tree, it also serves as factory for new nodes. Most of the other DDOM node classes actually do not have a public constructors. The DParser on top of the structure uses its functionality to create the desired DOM tree. Internally it uses
5. Implementation

a **SAXParser** to do most of the work. This will be explained later in this chapter. The parser is encapsulated by the two factory methods **DDocumentBuilder** and **DDocumentBuilderFactory** which provide the JAXP integration, allowing applications to select this or another DOM implementation by simply setting a system property. Figure 5.2 shows the hierarchy of the developed classes.

### 5.2 Dictionaries and structure array

The dictionary has to perform two main functions: Adding an entry to the dictionary and returning an index to it and returning an entry to a given index. Although this may look straightforward in the beginning it hides one important fact: Entries that already exist in the dictionary should not be appended again. This is the idea the whole compression is based on: avoidance of redundancy. If an entry already exists, the add function must return the index of it rather than appending the same data again. But before this can be performed, the dictionary must be able to find a certain entry.

Algorithms for searching are well understood. The runtime requirements of some algorithms on an ordered set of data are $\log(n)$, but this represents a problem. If the dictionary needs to be ordered at any given time, the position of the entries will change, invalidating references held by elements already added to the structure. The easiest way to avoid such problems is to use a linear search algorithm and an unordered data structure instead. A new entry is compared with every existing entry of the dictionary in turn and appended to the end if no such entry was found. This can become
5. Implementation

Fig. 5.1: The architecture of the developed DDOM system. Not all classes are shown for simplicity.
extremely slow when searching in large dictionaries. Nevertheless this is the way the dictionaries are implemented currently. Other possible solution to this problem are discussed in section 9.2. As the project was focused on the memory requirements of the implementation and not on its runtime performance, such issues were ignored for the time being.

Another question arises on how to implement the array containing the document structure. There are 14 different node types in the DOM. Now some of the node types will need more than one code, e.g. the element node needs one entry to mark the beginning and one to mark the end of it. The same applies for all nodes that can have children. For this reasons it was decided to use a byte to represent these codes, with constants declared for every possible entry type. It gives plenty of space for future extensions and is the smallest elementary Java type. The size required for the index
field referring to an entry in one of the dictionary tables is harder to find. Because most of the data structures used are arrays, it does not make sense to have something bigger than an integer, i.e. a 4-bytes-long, signed number, as array indices are limited to that size in Java. This limits the number of entries to $2^{31} \approx 2 \cdot 10^9$ entries, sufficient for all documents that can be kept in memory. For small documents this may seem a bit wasteful, but again the savings achieved by the use of a minimal bit-pattern, are outweighed by the increased memory usage caused by the use of an object type required to manage such a solution. However, the structure array itself is kept as small as possible. It starts with a reasonable small size and grows dynamically in case all existing entries are filled.

5.3 Improving performance for query relevant methods

Performance advantages are expected to come from the fact that most documents will fit into the computers main memory rather than being loaded from virtual memory, i.e. a hard disk. Thus the runtime behaviour of individual methods played only a minor role. But at least the performance of method calls relevant for querying large DOM trees should be kept in mind. There is one method, that is especially supplied for the convenience of querying: \texttt{Element.getElementsByTagName()}. This method allows the retrieval of all element nodes in the sub-tree of a given element node, which have a common name. The method allows the use of a wild-card ‘*’ to retrieve all elements in the sub-tree.

Of course it would be possible to traverse the tree, comparing the name
of element nodes with the given one. But it is far faster just to look-up the index of the given name in the dictionary and then to look for this index within the document structure. No nodes need to be created to traverse the tree, no Strings are compared. Only the nodes that are part of the search result are finally created and returned.

The method is based on a underlying, private method of the DocumentFragment object maintaining the data. The methods getEntries() and getAllChildren() return the indices of the entries and children in its array. This allows fast access and comparison without the need to create any DOM nodes internally. The code for the methods is shown in appendix C.

5.4 The modes of the nodes

As described in the previous chapter, every node can have two fundamentally different modes: An newly created, not appended node needs to contain all the information about itself, i.e. the data contained, its name and so one. But a node that is already appended to the structure, should only hold a minimal set of data, a reference to the object holding the data and a reference to its position within this structure. One of the major questions to tackle was whether these two roles could be fulfilled using just one kind of object.

If not, the interface for every type of node would have to be implemented twice: Once for the unconnected, heavyweight version, once for the appended lightweight version. Although this approach is technically feasible, it would nearly double the amount of objects that require to be implemented, increasing the development time and leading to some redundant implementations
Implementing the same functionality.

Defining one class that can fulfill both requirements is likely to have a bigger memory footprint though. However, the memory used by individual nodes will be of lower importance than one might expect. Because the design is based on the idea that nodes do not persist for longer than needed to append them to a document or to retrieve their data, only the data they store in the dictionary and array containing the structure will permanently consume memory, while memory used for the object representation is assumed to be released whenever possible. Nevertheless there are situations where a large number of objects may be returned by a query. Thus memory efficient coding of the nodes will be important as well.

It was decided to implement only one object per node type, which can represent both modes. One additional boolean variable was used to indicate the state of the node. Some variables will only be needed in the unconnected state. All these additional attributes like the textual data contained in a DText node increase the memory requirements. But because all of this user data is stored in String objects, only a reference is stored inside the node anyway. In case the string is no longer needed, the reference can be set to null, requiring only minimal space, while the String itself can be garbage collected.

Some other instance variables are used for different purposes in the two different states. A newly created node for example will always have a reference to its owning document, as it is its only connection with it and will be needed to fulfill some of the functionality. Once the node is appended to another node, say an element node, it is more important to store a reference
to this object, as it is maintaining the data of the node now. Of course the node will still need a possibility to retrieve a reference to the owning document, but this problem can now be referred to the parent node, i.e. it does not need to be stored internally.

Another example for this re-use of instance variables is the variable `index` of the base `DNode` class. If an element node is not appended to a parent node, this field will hold the index of the nodes name in the central dictionary. If it is appended it will hold the reference to this nodes entry in the parental node, which in turn can be used to retrieve the index of the nodes name in the dictionary.

In fact these two attributes, `owner` and `index`, together with the attribute indicating the nodes state, `appended`, are the only instance variables occurring in every node. Because all of the attributes used in the connected mode of the node are re-used in the unconnected version, the memory requirements for this combined mode are not bigger than they would be for a object that can only fulfill this single function. Only when used in its connected mode, some memory is wasted. In practice this difference is only a reference to a single `String`, making it negligible in comparison to the extra expense that would be required to implement two different classes for each of the node types.

However, one problem with the chosen implementation remains. This kind of re-use of resources can be very dangerous, as the content of instance variables has different meaning and require different handling in the different states. For this reason, all the instance variables are private and can only be accessed directly by their defining object, the class `DNode`. Even inherited
objects can set and get their values only by provided methods, which in turn must be made safe.

5.5 Parsing and JAXP integration

Although the WC3 Recommendation for the DOM [20] does not include a standardised way of parsing, a DOM implementation without a supplied parser is of limited use. DOM trees would need to be build from scratch, which would hardly comply with the initial design goals and declared usage scenarios. But building a parser from scratch is hard work. How can the effort spent on it be kept small without losing functionality?

The idea here is the re-use of already existing technology. Why invent the wheel twice? There are numerous parsers around which can do the job just fine. They just need to use the developed data structure instead of their own one. This is the point where SAX is useful. Rather than generating a structure in memory it triggers specified events as it parses a document. Building an event handler that actually catches these events and generates suitable calls to generate a DDOM tree is relatively easy.

The SAX parser used to do the main work of parsing is the SAX parser included in the Xerces [17] package. It does all the complicated tasks, like resolving entities, validating the structure using a DTD if one is present and identifying the different elements of the XML document. Although it would be possible to generate the array and dictionaries used by the DDOM directly, it is less error-prone and easier just to translate the SAX events into corresponding calls on the DOM API. Whenever the SAX parser reports the
beginning of an element, a new element node is created. All following nodes are appended to it until the corresponding end of the element is reported. Only then the element is closed and reported to its parent node. To avoid the depth of recursion, the list of open elements is stored on a stack, thus only allowing access to the current node.

Because of the lack of standardisation for parsing in the DOM standard, the developed parser is then wrapped into a factory class DDocumentBuilderFactory, which again is wrapped into a factory class called DDocumentBuilderFactory. This follows the standards defined by the JAXP and allows easy integration of the DDOM into JAXP compliant applications. All that is necessary to modify a XML application in order to use the DDOM is to set the system property `javax.xml.parser.DOMBuilderFactory` to point to the `DDocumentBuilderFactory` class.
6. TESTING

Testing the implementation of an abstract data type is not easy. There was no application developed which could have been tested. Testing in this case rather means to try to parse as many different types of XML documents as possible to ensure the implementation can handle them all. Once the XML file is successfully parsed into the DDOM architecture, the method calls provided for the different node types have to be called and their returned values need to be validated.

6.1 Test data

In order to test the capabilities of the implementation, mainly the two example file containing some data about a student shown in listing 2.1 and the domain name server database with different amount of server entries described in the following chapter were used. They represent two extreme cases of XML. The first one is a very short file, it is completely irregular and also contains a lot of less frequently used content elements such as attributes and entity references. It was used to prove that the implementation can handle all these exceptions properly. The second file has a fairly simple structure as it was generated automatically from a database. Thus its content is regular, containing a lot of redundant data. It is an ideal sample to demonstrate
the compression capabilities. This file was used two indicate the amount of compression achievable on real-world data and also to prove that the implementation can handle large volumes of data. Most real XML files should lie somewhere in between these two extremes.

6.2 Testing the internal structure of the implementation

The class DDocumentFragment has a method called dump(), which is also inherited by the DDocument. Calls on this method allow the data contained in their internal arrays to be written out to a specified PrintWriter, e.g. to a FileWriter enabling output to a file. These method calls were used during the testing phase to verify that the content of these arrays conforms with the design idea as shown in figure 4.2. The method also dumps the content of the dictionaries. A typical output from this method call is shown in listing A.1 in appendix A.

6.3 Testing the logical structure of the DOM tree

The class DNode, the parent class of all other nodes also has a dump() method. This function calls upon most of the public functions available for all nodes and prints the result. The method can be used recursively to gather information about all nodes of a (sub-) tree. However this is somehow restricted to small documents, as is becomes infeasible to verify the amount of data produced by a larger document.

A more visual approach to verify the structure of documents of any size is a small program taken from [4]. It allows the representation of the content
of a DOM tree within a JTree object of the Java Swing library. In this way it enables easy transversal of the tree using a graphical user interface (GUI). Figure 2.1 shows this program running with a fully expanded JTree for the student file. All nodes appear at their correct position and with correct values.

6.4 Other tests

Other methods were tested individually during the incremental development process. Methods newly implemented were tested on the described data sets to validate their output.
7. EXAMPLE DATA

In order to test the developed system, sets of real world data conforming to the initial assumptions were needed. The XML documents needed to have a certain size to allow fair measurements, although some small documents were included for completeness.

7.1 Domain name server database

A text file containing the server associated with a given domain name and its IP address was used to fill a relational database. The text file and the resulting table in the database are structured in the following way. There is an entry for the servers name, six different fields for the individual parts of the domain name and four fields for the components of the IP address. Because most domain names do not consist of six parts, a number of these fields have no value. The list was taken from the root domain name server for the “ac.uk” domain and thus contains only entries from it. That means that the first two fields used to store the domain name always contain the strings “uk” and “ac”.

The data from the relation is then read by a program and converted to an XML text file. The root element has the name SERVER-LIST. It contains one child named SERVER per tuple of the relation. These children in turn
have one child per field. The names of these elements are taken from the
column names. Within these elements the value of the field is stored using
a single text element. Fields containing null values are not used, i.e. the
program will create neither an empty text node nor the containing element.
This makes use of the fact, that an XML file does not have to be regular.

To allow different measurements, there are files with different cardinality
ranging from 100 to 100000 entries. The sizes of the corresponding XML
text files range from 21KB to 21MB respectively. Three slightly different
variants exist for each of these files. Variant one complies with typical format
conventions for XML data but does not refer to a DTD. Variant contains n
additional reference to a suitable, external DTD in its prolog. In the last
variant all white-spaces used to format the output are suppressed, making
it smaller in size but harder to read. Listing 7.1 shows a file containing just
one server entry in the first of the three formats.

7.2 Shakespearean plays

Jon Bosak has marked all of Shakespeare’s plays with mark-up language
and published them on the Internet. The XML documents, together with a
document type definition are available from [2]. The plays are structured in
a way typical for such documents, the DTD declaring this structure is shown
in listing 7.2. The content is divided into acts, scenes, speeches and lines at
the lowest levels. Additional some information about settings, persons and
stage directions are included using different elements. Listing 7.3 shows an
excerpt from the play “The Tragedy of Macbeth”.

Listing 7.1: An entry from the domain name server database stored in XML format

The XML version of a play has a typical length of about 150 – 200 KB. Obviously LINE elements are the most frequent ones and the content is pure text. The documents contain only element and text nodes. None of the elements has an attribute. Certain elements can appear at different contexts in the document. The element TITLE for example occurs as child of PLAY, ACT and SCENE. These files contain only minimal redundancy in the textual part. From the 2385 LINE elements in the play “Macbeth” 2373 are distinct, i.e. only 12 strings do not need to be stored again. Nearly all the compression occurring on this kind of document must come from the compact representation of the mark-up structure.
<ENTITY amp "&#38;:">  
<!ELEMENT PLA Y (TITLE, FM, PERSONAE, SCNDESCR, PLA YSUBT, INDUCT?, PROLOGUE?, ACT+, EPILOGUE?)>  
<!ELEMENT TITLE (#PCDATA)>  
<!ELEMENT FM (P+)>  
<!ELEMENT P (#PCDATA)>  
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>  
<!ELEMENT PGROUP (PERSONA+, GRPDESCR)>  
<!ELEMENT PERSONA (#PCDATA)>  
<!ELEMENT GRPDESCR (#PCDATA)>  
<!ELEMENT SCNDESCR (#PCDATA)>  
<!ELEMENT PLA YSUBT (#PCDATA)>  
<!ELEMENT INDUCT (TITLE, SUBTITLE∗, (SCENE+|(SPEECH|ST AGEDIR|SUBHEAD)+))>  
<!ELEMENT ACT (TITLE, SUBTITLE∗, PROLOGUE?, SCENE+, EPILOGUE?)>  
<!ELEMENT SCENE (TITLE, SUBTITLE∗, (SPEECH | ST AGEDIR | SUBHEAD)+)>  
<!ELEMENT PROLOGUE (TITLE, SUBTITLE∗, (ST AGEDIR | SPEECH)+)>  
<!ELEMENT EPILOGUE (TITLE, SUBTITLE∗, (ST AGEDIR | SPEECH)+)>  
<!ELEMENT SPEECH (SPEAKER+, (LINE | ST AGEDIR | SUBHEAD)+)>  
<!ELEMENT SPEAKER (#PCDATA)>  
<!ELEMENT LINE (#PCDATA | ST AGEDIR)∗>  
<!ELEMENT ST AGEDIR (#PCDATA)>  
<!ELEMENT SUBTITLE (#PCDATA)>  
<!ELEMENT SUBHEAD (#PCDATA)>  

**Listing 7.2:** The document type definition for the Shakespearean plays

```
<ACT><TITLE>ACT I</TITLE>

<SCENE><TITLE>SCENE I. A desert place.</TITLE>
<ST AGEDIR>Thunder and lightning. Enter three Witches</ST AGEDIR>

<SPEECH>
<SPEAKER>First Witch</SPEAKER>
<LINE>When shall we three meet again</LINE>
<LINE>In thunder, lightning, or in rain?</LINE>
</SPEECH>
```

**Listing 7.3:** An excerpt from the “The Tragedy of Macbeth” in XML
8. MEASUREMENTS

This chapter describes how the memory measurements were performed and their outcome. In order to allow comparisons, the measurements were repeated on two other popular DOM implementations, Xerces [17] and Crimson [16], both of which do not use any form of internal compression.

8.1 Measurement procedure

Java does not allow much insight into its memory management. This is deliberate. It stops programmers manipulating memory directly, the source of much trouble in other programming languages. There is no information about the memory usage of single objects available. The only quantities that can be measured are the total memory of the virtual machine and the remaining free memory. This at least allows an estimate of the memory being used at run time, although certain limitations apply.

The memory reported as free will depend on the success and progress of the garbage collector. Again only little control over the garbage collector is given to the programmer. By calling a method of the object representing the runtime environment, an application program can suggest that the garbage collection routine is run. It is not guaranteed though. Another uncertainty factor is the degree to which obsolete objects are detected and discarded.
Using different implementations of the Java runtime machine and its class library shows significant differences in terms of performance and memory requirements of an application program. To eliminate their influence, all measurements were performed using the same Java environment, namely Sun’s Java Development Kit version 1.3.1.

To ensure comparability all measurements were performed in the following way. The test program starts and does all preparations before the first memory measurement is taken. The algorithm used to do this is shown in figure 8.1. It calls upon the garbage collector before performing any measurements. It then calculates the amount of memory used as difference between reported total and free memory of the virtual Java machine. After this is done the XML document in question is parsed into a DOM tree. Afterwards the memory measurement is repeated and the difference is assumed to correspond to the memory used by the DOM tree. This does not need to be exact. The garbage collector might have left unused objects behind or the memory occupied by objects already present before the initial measurement took place may have been used up during the construction of the DOM tree. Errors in both directions are possible, but assumed to be small compared to the memory consumed by large documents.

To keep these effects to a minimum, subsequent creation and garbage collection of objects resulting in memory fragmentation should be avoided. Experiments have shown that the same document parsed twice in a single run of the test program does not necessarily result in the same memory usage being reported. For this reason, only one document and only one DOM implementation are tested during any one run of the test program.
8. Measurements

```java
public static long getUsedMemory() {
    Runtime rt = Runtime.getRuntime();
    rt.gc();
    return rt.totalMemory() - rt.freeMemory();
}
```

*Listing 8.1: The memory measurement routine*

However, runs using identical DOM implementations and XML documents in the same Java environment result in exactly the same memory usage, i.e. the measurements are repeatable.

## 8.2 Domain name database

The domain name database contains a lot of redundant data and should therefore give an upper limit of the performance of the compression algorithm. Remembering the fact that two of the fields in the original database, the two most significant parts of the domain name, have the constant values “uk” and “ac”, a relatively high compression rate is expected. The values contained in the other fields should also allow reasonable savings. The four entries for the components of the IP address for example can only range from 0 to 255. In a table with 10,000 entries most of these values will occur several times. Only the first field, the server name, is nearly unique for each individual entry, resulting in a dictionary for this domain that has nearly the same cardinality as the the original table.
Table 8.1: The memory usage of different DOM implementations for the domain name database compared to the size of the XML file. Notes: \(^1\)relative to the DDOM \(^2\)with manually increased heap

### 8.2.1 The influence of the document size

Table 8.1 and figure 8.1 show the memory usage for different volumes of data measured in the way described above. The measurements were performed in absence of a DTD, i.e. all white-space used for indentation will be included in the resulting DOM tree. Although the compression of the DDOM implementation is significant compared with the other DOM implementations, it still lies above the file-size of the XML source document. Only for large documents the DDOM implementation can approach this figure. Note that only Crimson’s memory usage is linear, improving its otherwise bad performance for small documents. While the overhead of the DDOM implementation for small documents was expected as dictionaries need to be maintained and the compression algorithms can only do little on limited data, the increase of memory usage per entry using the Xerces parser is surprising. It was assumed to be an artifact caused by memory segmentation during the document creation.
8. Measurements

**Fig. 8.1:** Memory consumption of the different DOM implementations for the domain name server database

### 8.2.2 The influence of the document style

The above measurements were performed without using the available DTD to enable validating. The XML files were formatted following existing conventions. The hierarchy of the document was indicated by indentation. Elements which only have a single text child are printed in one line including the textual content. Element tags enclosing a more complicating structure are printed on separate lines. The examples shown in listing 2.1 shown this structure.

This way of noting an XML document is used to support readability. Nevertheless it must be stated here, that the additional line feed, white-spaces and carriage return characters included are content as well. Without a DTD
a parser can not distinguish between white space used to indicate the structure of the document and a text field containing white-spaces. Obviously these additional nodes in the corresponding DOM tree are unwanted. There are two possible approaches to get rid of them. Using a DTD allows the designer to indicate the positions where character data is supposed to occur. For the student example the NAME element is declared to have only the two elements SURNAME and FIRSTNAME as children, all the white spaces between the individual element tags are therefore recognised as such. Nevertheless they are included by default, and can only be excluded by setting a property of the DocumentBuilderFactory class. This was done for the measurements performed in presence of the DTD. Without this setting, the results are nearly exactly the same as in the case without DTD. Another approach is not to include such unwanted content in the first place. XML documents do not have to follow the conventions described above to remain well-formed or valid. Writing the entire document in a single line, without any additional white-spaces is legal and possible, although it contradicts with the goals of readability and clarity.

Obviously the way a document is formatted will have an impact on the DOM tree. As additional nodes need to be included if white-space is present, memory requirements will increase. To demonstrate the effect of the XML format on the different implementations, the XML document containing the 10,000 entries from the domain name database where tested in the three described formats.

Table 8.2 shows the result for these measurements. Crimson could not be used for the validating run as it always raises a runtime exceptions on
documents referring to an external DTD. Whether this results from a configuration error or is a general limitation of this implementation could not be identified. As work on its code-base is practically stopped in favour of a new version of Xerces, this seemed to be of minor importance.

Note that the comparison is not fair for Xerces as well, as the parser developed for the DDOM implementation currently does not handle document type information. Nevertheless, the data stored in the document type node of a DOM document contains default attribute values, notations and entity declarations only. As none of these are present in the domain name server database it seemed unlikely that it will increase DDOM’s memory consumption by more than a fraction.

The results follow the expectations. Because the indentation used throughout the document usually remains constant for a given element, only few additional entries in the dictionaries are needed. Only the array containing the document structure of a DDOM document will grow by one entry per additional text node. The overall result for this implementation is negligible. In fact the additional objects created during the parsing process increase the memory usage for the validated document slightly. For Xerces things look different. It can benefit from the compact document representations as a

<table>
<thead>
<tr>
<th>Format</th>
<th>XML file</th>
<th>DDOM (rel.)</th>
<th>Xerces (rel.)</th>
<th>Crimson (rel.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard using DTD</td>
<td>2.1 MB</td>
<td>3.24 MB (1.0)</td>
<td>14.85 MB (4.6)</td>
<td>19.31 MB (6.0)</td>
</tr>
<tr>
<td>compact</td>
<td>1.6 MB</td>
<td>3.19 MB (1.0)</td>
<td>9.07 MB (2.8)</td>
<td>12.85 MB (4.0)</td>
</tr>
</tbody>
</table>

Tab. 8.2: The memory usage of different DOM implementations for the domain name database formatted using different styles. Notes: \(^1\)relative to the DDOM \(^2\)caused runtime exception
lot less nodes need to be stored. The same applies for the run with validating DTD and suppression of white-space activated. Without this setting, the result remains nearly the same as for the XML file without DTD (not shown).

8.2.3 Comparison with relational data structures

As the project was aimed at nearly regular data structures, a comparison with a relational database system was of interest. The data of the domain name server database already has a relational structure, although it is stored in a text file. Loading this data into different database systems gives an estimate on memory requirements of these very different systems.

However, it should be noted here, that the comparison is not necessary fair for either side. The data is simply loaded into a single big relation. One the one hand there are more efficient ways to represent the given data in a database. One could put the reoccuring entities into separate tables and use joins to reduce the redundancy. But on the other hand XML will offer more flexibility. With XML one does not have to decide in advance, how long a single entry can become. With databases this is crucial. If data would arise in the future, that would not fit into the relational structure chosen, XML will show its extensibility. The database also has the advantage of having more primitive types. In case of the test data, it will be able to store the integers from the IP address as such and not in textual form.

10,000 entries of the test data were converted to a different file formats. It was converted to a comma separated text file first and than imported
8. Measurements

<table>
<thead>
<tr>
<th>Format</th>
<th>Description/Comments</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSV</td>
<td>Fields separated by commas, values enclosed in quotes one entry per line</td>
<td>475.5 KB</td>
</tr>
<tr>
<td>dBase</td>
<td>One relation, no primary key, IP address stored as number, generated by StarOffice</td>
<td>908.6 KB</td>
</tr>
<tr>
<td>MS Access</td>
<td>One relation, no primary key, one index on server names, IP address stored as number</td>
<td>696.0 KB</td>
</tr>
<tr>
<td>XML file</td>
<td>No DTD, conventional formatting</td>
<td>2,205.8 KB</td>
</tr>
<tr>
<td>DDOM tree</td>
<td>No validation, includes white-space</td>
<td>3,242.7 KB</td>
</tr>
</tbody>
</table>

**Tab. 8.3:** The memory usage for 10,000 entries of the domain name server database stored in different formats

into Microsoft Access and Sun’s StarOffice. The later program was used to generate a dBase compatible file. The size of each of these files is shown in table 8.3. For comparison, the file size of the XML file in standard format and the memory usage of the DDOM implementation for this file is also given.

All of the database formats require less storage space than the XML or DOM representation. This is mainly caused by the use of suitable encoding formats, i.e. integers to store the IP address and 8 bit encoded characters to store the textual values. Differences between different strategies become obvious. Although dBase uses a relatively simple file format, its files are biggest under the tested databases as it stores text fields always in strings with the maximal allowed length. Compared to the comma separated file format much space is wasted as most of the domain name parts are quite short, but need to be stored in a string with the length of the longest domain name.
8. Measurements

<table>
<thead>
<tr>
<th>Play</th>
<th>XML file</th>
<th>DDOM (rel.)¹</th>
<th>Xerces (rel.)¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macbeth</td>
<td>159.3 KB</td>
<td>628.2 KB (1.00)</td>
<td>870.1 KB (1.39)</td>
</tr>
<tr>
<td>Julius Caesar</td>
<td>179.2 KB</td>
<td>739.6 KB (1.00)</td>
<td>981.6 KB (1.33)</td>
</tr>
<tr>
<td>Henry VIII</td>
<td>212.7 KB</td>
<td>825.6 KB (1.00)</td>
<td>1092.8 KB (1.32)</td>
</tr>
</tbody>
</table>

Tab. 8.4: The memory usage of DDOM and Xerces parser for some of Shakespeare’s plays. Note: ¹relative to the DDOM

8.3 Shakespearean plays

Three of Shakespeare’s plays were parsed by the different DOM implementations as described above. The results are shown in table 8.4. Note that the compression measured must come from the compact representation of the mark-up structure as nearly all textual data of the documents is distinct. Although the documents are relatively small compared with the domain name database, it gives a good indication of how much compression can be achieved on documents with only few redundant entries but a large number of re-occurring mark-up tags. The measurements were performed using JAXP standards, i.e. validation was used but white space is included in the DOM tree. No measurements could be performed using the Crimson parser as it failed with an exception during any try to parse an document with an external DTD.
9. CONCLUSIONS

9.1 Achievements

The DDOM implementation produces documents that are on average significantly smaller than those generated by competing DOM implementations. This becomes obvious for highly regular and redundant data structures as typical for large, data-centric XML files. It therefore gives a better chance of processing large XML documents without using external storage. Not all of the functionality as defined by the W3C could be implemented in the given time. Notations are not implemented by the DDOM and entities and their references are only handled properly if the underlying SAX parser used can resolve them. The same applies to document type definitions. Simple attributes, i.e. attributes containing a single text value are handled correctly, entities within attributes need to be resolved by the SAX parser. Default values and references are currently ignored by the DDOM. Despite these limitations all of the example documents could be parsed correctly. This shows, that a majority of real-world XML documents do not make use of all the features provided by XML. Simple programs as described in [11] run on the DDOM implementation without change. This reassures about the standard consistency of the developed solution.
9.2 Suggestions for future work

9.2.1 Improvements on compression ratio

Although significant compression compared to conventional DOM implementations could be achieved, the parsed documents still occupy more memory than the original textual version of the XML documents. Considering the redundancy present in the text version, this seems to be unsatisfying. Reasons for this increase needs to be investigated to allow more efficient compression algorithms to be developed. It seems to be crucial to use additional information supplied by DTDs. However, in the case of Java as an implementation language, the biggest waste seems to be the expensive representation of strings using 16bit unicode characters. The emerging XML Schema standard [23] gives reason for optimism. If data in XML documents will be strongly typed in the future, more efficient internal representations could be used.

9.2.2 Improvements on dictionary and parser implementation

The dictionary structure implemented uses a linear search algorithm to check whether or not an entry already exists in it. This is an unnecessary waste of processing time, as it could be improved using a different internal organisation. There are two possible solutions to this problem:

Firstly it would be possible to use some kind of internal mapping. The dictionary itself could be stored in a sorted way, but the indices used externally would be mapped to the real indices of the dictionary. Thus it would
be possible to maintain validity of already returned indices whilst changes on the internal structure become allowable. Of course this introduces other problems: The mapping would require an additional layer between the dictionary and the DDOM nodes, increasing memory requirements and slowing performance.

Another, far easier approach is to construct a two-pass parser. The first pass would build up the dictionaries, which are subsequently sorted. The next pass generated the DDOM nodes. Because all the entries required already exist in the dictionaries, no new entries needs to be added at this stage, maintaining the sorted structure.

The second approach could be tackled together with another improvement. So far the document is parsed using a SAX parser that is out with the control of the program. The DDOM parser only works successfully if a default SAX parser is available on the machine. If the parser installed is erroneous or incomplete, the DDOM structure build will be erroneous as well. One example of such an effect is the handling of comments in the XML code. SAX 1.0 parsers do not generate events when they come across comments. Thus comments are never included in a document generated by the current parser. Although this is consistent with the recommendations of the W3C it is worrying that such features are currently out with the control of the used parser.
9.2.3 Extending the functionality

While some of the limitations of the DDOM are fundamental, others seem to be a question of a better implementation. There is no need to lock every node for writing once it is appended to a parent. Insertion, replacement and removal methods could be implemented, although some of these functions would result in poor runtime behaviour. The main problem with these methods is to make sure, that eventually existing nodes referring to other already appended data remain valid.

Additional support for DOM level 2 [21], especially support for namespaces, would help to make the implementation fit for the future.
References

[1] Craig A. Berry, Helen Callaghan, Jim Molony, Steve Rycroft, and Chan-
noch Wiggers, editors. Professional Java XML. Wrox Press Ltd., Birm-

ibiblio.org/xml/examples/shakespeare/.

[3] W. Paul Cockshot, Douglas McGregor, and John Wilson. High-
performance operations using a compressed database architecture. The

[1], 2001.


representation and exchange of xml over the web. In Proceedings of the

Birmingham, 1999.


A. DEBUGGING OUTPUT FROM DDOCUMENTFRAGMENT.DUMP()

The following listing shows the output produced by the `DDocumentFragment.dump()` method. Because the example file `student.xml` has a DTD, white spaces where already dismissed during the parsing process and do not occur in the dictionaries. Note also that the entity reference used to identify the umlaut '¨ u' in the student’s surname was resolved by the parser.
Dictionary for type 2, domain 0:

1: STUDENT
2: NAME
3: SURNAME
4: FIRSTNAME
5: COURSE
6: DEGREE

Dictionary for type 4, domain 1:

1: 200053493

Dictionary for type 4, domain 3:

1: Neumüller

Dictionary for type 4, domain 4:

1: Mathias

Dictionary for type 4, domain 5:

1: Information Technology Systems

Dictionary for type 4, domain 6:

1: Master of Science

Dictionary for type 7, domain 1:

1: REGNO

*Listing A.1: Debug output for the student example*
B. API DOCUMENTATION

The API documentation for the implemented classes was generated using the Javadoc tool \cite{15}. This generates a HTML hypertext document from specially formatted documentation comments within the source code. Due to its format, the documentation is of limited use in printed form. A hypertext version of the documentation can be found on the disk accompanying this report or at \url{http://www.cs.strath.ac.uk/~mneumuel/ddom/index.html}. 
C. SOURCE CODE

The following contains the code written for the DDOM implementation. Note that comments in the source code only appear at the highest level in the class hierarchy. Functionality provided by overloaded methods is described in the class or interface where the method is declared. Only subsequent changes are documented in overloaded versions. To allow easy reading of those comments, the hypertext version of the API documentation provided is more suitable. This is of special importance for the functionality defined by the DOM interfaces. As the requirements for some of the methods are quite complicated, these are not repeated in the DDOM implementation, but should be looked up in the relevant interface definitions.

DOM data structure (package ddom.dom)

Class ddom.dom.DNode

```java
package ddom.dom;

import org.w3c.dom.*;
import java.io.PrintWriter;

/**
 * The abstract class DNode represents a single node in a DDOM tree.
 * This class is the super class of all other node types in the DDOM.
 * It defines several methods for traversing and modifying the DOM tree although
 * the usefulness of some of these methods will depend on the current type of
 * the node. For details on nodes check the documentation of the
 * Node interface. */
 *
 * @see org.w3c.dom.Node
 * @author Mathias Neumüller
 * @version 1.0
 */
abstract public class DNode implements Node {
    // declaration of constants
    *
    * These constants are used in the internal type[] areas to indicate the type
    * of a certain entry.
    */
    static final byte STARTDOCUMENTENTRY = 1;
```
static final byte END_DOCUMENT_ENTRY = -1;
static final byte START_ELEMENT_ENTRY = 2;
static final byte END_ELEMENT_ENTRY = -2;
static final byte START_DOCUMENT_FRAGMENT_ENTRY = 3;
static final byte END_DOCUMENT_FRAGMENT_ENTRY = -3;
static final byte TEXT_ENTRY = 4;
static final byte COMMENT_ENTRY = 5;
static final byte CDATA_SECTION_ENTRY = 6;
static final byte START_ATTRIBUTE_ENTRY = 7;
static final byte END_ATTRIBUTE_ENTRY = -7;
static final byte ENTITY_REFERENCE_ENTRY = 8;
static final byte ENTITY_ENTRY = 9;
static final byte NOTATION_ENTRY = 10;
static final byte PROCESSING_INSTRUCTION_ENTRY = 11;
static final byte START_DOCUMENT_TYPE_ENTRY = 12;
static final byte END_DOCUMENT_TYPE_ENTRY = -12;

/*
 * These constants are used in the internal data[] areas or for the index
 * field to indicate some special conditions.
 */
static final int NO_DOMAIN = 0;
static final int NO_DATA = -1;
static final int NOT_IN_TREE = -2;
static final int ALL_DATA = -3;
static final int NO_ENTRY = -4;

/**
 * Used to separate different content in some types of nodes,
 * e.g. DProcessingInstruction.
 */
static final char SEPARATOR = ' ';

// declaration of instance variables

/**
 * This variable is used to indicate whether this node is appended to a parent
 * node, or if it still unconnected, i.e. can be changed.
 */
private boolean appended;

/**
 * This variable holds either the index to this element in the structure,
 * if it was appended, or, if possible, the index to its entry in the
 * dictionary, if it was not appended yet but is data is entered into the
 * dictionary.
 */
private int index;

/**
 * This variable hold either a reference to the document itself as long as
 * the node is not appended to a parent. If the node is appended to a parent
 * node, it holds a reference to the document fragment that is storing the
 * details about this node.
 */
private DDocumentFragment owner;

// constructor methods

/**
 * Constructor that creates a reference node. This constructor can only be
* called by constructor methods of inherited classes as this class is
* abstract.
*/
DNode(DDocumentFragment owner, int index) {
    this.owner = owner;
    setEntryIndex(index);
    appended = (index != NOT_IN_TREE);
}
/**
 * Constructor that creates a new, unconnected node. This constructor can
 * only be called by constructor methods of inherited classes as this class
 * is abstract.
 */
DNode(DDocument owner) {
    this(owner, NOT_IN_TREE);
}

// implementation of Node methods
public String getNodeName() { return null; }
/**
 * The value of this node, depending on its type.
 * Nodes that do not hold a specific value will return "null".
 * @exception org.w3c.dom.DOMException DOMSTRING_SIZE_ERR: Raised when it
 * would return more characters than fit in a DOMString variable on the
 * implementation platform.
 */
public String getNodeValue() throws DOMException {
    return null;
}
/**
 * Sets the value of this node to the specified value.
 * This operation makes only sense for certain types of nodes.
 * Node which do not support this method will throw an exception.
 * @exception org.w3c.dom.DOMException NO_DATA_ALLOWED_ERR If the node can
 * not hold data.
 * @exception org.w3c.dom.DOMException NO_MODIFICATION_ALLOWED_ERR: Raised
 * when the node is readonly.
 */
public void setNodeValue(String nodeValue) throws DOMException {
    throw new DDOMException(DOMException.NO_DATA_ALLOWED_ERR,
        "DNode_error: Nodes of this type can not hold data.");
}
public abstract short getNodeType();
public Node getParentNode() { if (isAppended()) {
   int pindex = getFragment().getParentEntry(getEntryIndex());
   if (pindex != NO_ENTRY) {
       return getFragment().createNode(pindex);
   } else {
       return null;
   }
} else {
}
public NodeList getChildNodes() {
    return new DNodeList();
}

public Node getFirstChild() {
    return null;
}

public Node getLastChild() {
    return null;
}

public Node getPreviousSibling() {
    if (isAppended()) {
        int index = getEntryIndex();
        int level = 0;
        boolean found = false;
        while ((level>=0) && !(found) && (index>0)) {
            index--;
            if (getFragment().isParentEntry(index)) level--;
            if (getFragment().isEndParentEntry(index)) level++;
            if ((level==0) && getFragment().isChildEntry(index)) found = true;
        }
        if (found) {
            return getFragment().createNode(index);
        } else {
            return null;
        }
    } else {
        return null;
    }
}

public Node getNextSibling() {
    if (isAppended()) {
        int index = getEntryIndex();
        int level = 0;
        if (getFragment().isParentEntry(index)) level++;
        boolean found = false;
        while ((level>=0) && !(found) && (index+1<getFragment().getSize())) {
            index++;
            if ((level==0) && getFragment().isChildEntry(index)) found = true;
            if (getFragment().isParentEntry(index)) level++;
            if (getFragment().isEndParentEntry(index)) level--;
        }
        if (found) {
            return getFragment().createNode(index);
        } else {
            return null;
        }
    } else {
        return null;
    }
}

public NamedNodeMap getAttributes() {
    return null;
}
public Document getOwnerDocument() {
    return getOwner();
}

/**
 * Inserts a child node before a given node already present in this node.
 * Please note that this method will always throw a DOM Exception in the DDOM
 * API. This is because the DDOM currently supports only the appendChild
 * method to modify the children contained in a node.
 * @exception DOMException NOT_SUPPORTED_ERR in the current version
 */
public Node insertBefore(Node newChild, Node refChild) throws DOMException {
    throw new DDOMException(DOMException.NOT_SUPPORTED_ERR,
        "DNode_error:Can_not_insert_children._use_appendChild().instead.");
}

/**
 * Replaces an already present child node in this node with a given node.
 * Please note that this method will always throw a DOM Exception in the DDOM
 * API. This is because the DDOM currently supports only the appendChild
 * method to modify the children contained in a node.
 * @exception DOMException NOT_SUPPORTED_ERR in the current version
 */
public Node replaceChild(Node newChild, Node oldChild) throws DOMException {
    throw new DDOMException(DOMException.NOT_SUPPORTED_ERR,
        "DNode_error:Can_not_replace_children.");
}

/**
 * Removes an already present child node from this node.
 * Please note that this method will always throw a DOM Exception in the DDOM
 * API. This is because the DDOM currently supports only the appendChild
 * method to modify the children contained in a node.
 * @exception DOMException NOT_SUPPORTED_ERR in the current version
 */
public Node removeChild(Node oldChild) throws DOMException {
    throw new DDOMException(DOMException.NOT_SUPPORTED_ERR,
        "DNode_error:Can_not_remove_children.");
}

public Node appendChild(Node newChild) throws DOMException {
    throw new DDOMException(DOMException.HIERARCHY_REQUEST_ERR,
        "DNode_error:Can_not_append_children_to_this_node_type(" +nodeName+".");
}

public boolean hasChildNodes() {
    return false;
}

/**
 * Returns the dictionary entry for the given type, domain and index.
 */
String getDictionaryEntry(byte type, int domain, int index) {
    return getOwner().getDictionary().get(type, domain, index);
}

/**
 * Returns the dictionary entry for the given type and index.
 */

String getDictionaryEntry(byte type, int index) {
    return getOwner().getDictionary().get(type, index);
}

/**
 * Returns a dictionary index for the given value of the given type and
 * domain.
 */
int appendDictionaryEntry(byte type, int domain, String value) {
    return getOwner().getDictionary().append(type, domain, value);
}

/**
 * Returns a dictionary index for the given value of the given type.
 */
int appendDictionaryEntry(byte type, String value) {
    return getOwner().getDictionary().append(type, value);
}

/**
 * Returns the type constant that is used to indicate the beginning of this
 * type of node in the type[] array of DDocumentFragments.
 */
abstract byte getEntryType();

/**
 * Returns the domain in which this node exists. This is either the constant
 * <CODE>NO_DOMAIN</CODE> for nodes whose values are not members of a certain
 * domain, e.g. Documents or Elements, or the entry index of the name of the
 * element node forming the domain. Nodes that are not yet appended to the
 * structure do not have a domain and thus return the <CODE>NO_DOMAIN</CODE>.>
 * An example of some nodes and their domains:
 * <CODE>&lt;ROOTELEMENT&gt;&lt;BR&gt;</CODE>
 * &lt;!-- AComment --&gt;&lt;BR&gt;
 * &lt;AELEMENT AAttr="AValue">AText</AELEMENT&gt;&lt;BR&gt;
 * &lt;/ROOTELEMENT&gt;&lt;BR&gt;
 * &lt;/AATTRIBUTE&gt;&lt;BR&gt;
 * The Document, ROOTELEMENT and AELEMENT have <CODE>NO_DOMAIN</CODE>.
 * AComment is in domain <VAR>ROOTELEMENT</VAR>.
 * Note that although the last three nodes exist in the same domain, they
 * are not necessary stored in the same dictionaries, as dictionaries are
 * divided by node types. Nevertheless, AValue is represented as Text as is
 * AText, thus these will be stored in the same dictionary.
 */
int getEntryDomain() {
    if (isAppended()) {
        return getFragment().getEntryDomain(getEntryIndex());
    } else {
        return NO_DOMAIN;
    }
}

/** Sets the index attribute of this node to the given value. */
void setEntryIndex(int index) {
    this.index = index;
}

/**
 * Returns the index attribute of this node.
 */
int getEntryIndex() {
    return index;
}
C. Source code

```java
/**
 * Returns the index of the data of this node in the corresponding dictionary.
 */
int getDataIndex() {
    if (isAppended()) {
        return getFragment().getEntryData(index);
    } else {
        return index;
    }
}

/**
 * Returns if the entry indicated by the given type can hold children.
 */
boolean isParentEntry(byte type) {
    switch (type) {
    case STARTELEMENTENTRY:
    case STARTDOCUMENTENTRY:
    case STARTDOCUMENTFRAGMENTENTRY:
    case STARTATTRIBUTEENTRY:
        return true;
    default:
        return false;
    }
}

/**
 * Returns if the entry indicated by the given type marks the ends of a
 * node that can hold children.
 */
boolean isEndParentEntry(byte type) {
    return (type < 0);
}

/**
 * Returns if the entry indicated by the given type can be a child node.
 */
boolean isChildEntry(byte type) {
    switch (type) {
    case STARTELEMENTENTRY:
    case TEXTENTRY:
    case COMMENTENTRY:
    case CDATASECTIONENTRY:
    case ENTITYREFERENCEENTRY:
    case PROCESSINGINSTRUCTIONENTRY:
        return true;
    default:
        return false;
    }
}

/**
 * Checks if the current node can be appended to the specified fragment.
 * This method is called prior to any operation that appends a new child node
 * to the tree structure. This method will throw an exception in case child
 * node and the fragment are part of different documents or if the child was
 * already appended elsewhere.
 * @exception org.w3c.dom.DOMException NO_MODIFICATION_ALLOWED_ERR if the node
 * was already appended elsewhere.
 * @exception org.w3c.dom.DOMException WRONG_DOCUMENT_ERR if the node was
 * belongs to a diferent document than the fragment.
 */
void checkValidity(DDocumentFragment fragment) throws DOMException {
    if (isAppended()) {
        throw new DOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
```
"DNode error: Nodes can only be appended to the DOM tree once only.");

if (fragment.getOwnerDocument() != getOwnerDocument()) {
    throw new DOMException(DOMException.WRONG_DOCUMENT_ERR,
        "DNode error: Nodes can only be appended to the document which was used to create them.");
}

/**
 * This method is called when this node is appended to another one. The child
 * node performs all the necessary actions to append itself to the specified
 * fragment. By default this will cause the child node to become read-only.
 */
DNode appendEntry(DDocumentFragment fragment, int domain) throws DOMException {
    owner = fragment;
    close();
    return null;
}

/**
 * Determines whether the node is appended to a parent node.
 */
boolean isAppended() {
    return appended;
}

/**
 * Opens the node, that is sets it to read-write mode.
 */
void open() {
    appended = false;
}

/**
 * Closes the node, that is sets it to read-only mode.
 */
void close() {
    appended = true;
}

/**
 * Returns the owning document of this node.
 */
DDocument getOwner() {
    if (isAppended()) {
        return getFragment().getOwner();
    } else {
        return (DDocument) owner;
    }
}

/**
 * Returns the containing DocumentFragment of this node.
 */
DDocumentFragment getFragment() {
    if (isAppended()) {
        return owner;
    } else {
        return null;
    }
}

/**
 * Prints the properties of the node.
 * This method is provided for debugging purposes and may be removed from
 * future versions. It returns the properties of this node in tabular form.
 * @param writer A PrintWriter to which the output is sent
 * @param recursive Defines whether the procedure is called recursively on all
 * of its children.
 */
public void dump(PrintWriter writer, boolean recursive) {
    writer.println("=".repeat(40));
    writer.println("Information about node: "+toString());
    writer.println("=".repeat(40));
    writer.println("Node name: "+getNodeName());
    writer.println("Node value: "+getNodeValue()).trim();
    writer.println("Node type: "+getNodeType());
    writer.println("Owner document: "+getOwnerDocument());
    writer.println("Parent node: "+getParentNode());
    writer.println("Previous sibling: "+getPreviousSibling());
    writer.println("Next sibling: "+getNextSibling());
    writer.println("Node reports children: "+hasChildNodes());
    writer.println("First child: "+getFirstChild());
    writer.println("Last child: "+getLastChild());
    writer.println();
    writer.flush();
    if (recursive) {
        DNodeList children = (DNodeList) getChildNodes();
        for (int i = 0; i < children.getLength(); i++) {
            ((DNode) children.item(i)).dump(writer, recursive);
        }
    }
}

/**
 * Returns a String representation of the node.
 * The returned String consists of the literal text "Node" plus the node name
 * in brackets, followed by the node value, separated by a colon.
 */
public String toString() {
    return ("Node (" + getNodeName() + ") : " + getNodeValue()).trim();
}

Class ddom.dom.DGroupNode

package ddom.dom;

import org.w3c.dom.*;

/**
 * The abstract class DGroupNode represents groups of nodes. It is the super
 * class of all nodes that are represented as a list of entries in the structure
 * array rather than as a single entry, such as documents, elements and
 * attributes. It adds one abstract method, <CODE>getEndEntryType()</CODE>,
 * that needs to be implemented by all children and provides
 * some helper methods to handle cloning and appending of NodeLists.
 * @author Mathias Neumüller
 * @version 1.0
 */
abstract public class DGroupNode extends DNode {
    // constructor methods
    DGroupNode(DDocumentFragment owner, int index) {
        super(owner, index);
    }
    // implementation of Node methods
public boolean hasChildNodes() {
    return (getFragment().getDirectChildren(getEntryIndex()).length > 0);
}

public NodeList getChildNodes() {
    int[] childrenref = getFragment().getDirectChildren(getEntryIndex());
    DNodeList children = new DNodeList();
    for (int i = 0; i < childrenref.length; i++) {
        children.append(getFragment().createNode(childrenref[i]));
    }
    return children;
}

public Node getFirstChild() {
    int[] children = getFragment().getDirectChildren(getEntryIndex());
    if (children.length > 0) {
        return getFragment().createNode(children[0]);
    } else {
        return null;
    }
}

public Node getLastChild() {
    int[] children = getFragment().getDirectChildren(getEntryIndex());
    if (children.length > 0) {
        return getFragment().createNode(children[children.length - 1]);
    } else {
        return null;
    }
}

// implementation of DGroupNode methods

abstract byte getEndEntryType();

/**
 * Appends all node from the given NodeList as children to this node.
 */
void appendNodeList(NodeList children) {
    for (int i = 0; i < children.getLength(); i++) {
        appendChild(children.item(i));
    }
}

/**
 * Returns a NodeList containing the cloned children of this node.
 * The returned children are cloned with the parameter deep set to true, i.e.
 * the children’s children will be cloned recursively.
 */
NodeList getClonedChildNodes() {
    NodeList children = getChildNodes();
    DNodeList clones = new DNodeList();
    for (int i = 0; i < children.getLength(); i++) {
        clones.append(children.item(i).cloneNode(true));
    }
    return clones;
}
Class `ddom.dom.DDocumentFragment`

```java
package ddom.dom;

import org.w3c.dom.∗;
/
∗∗
∗ The class DDocumentFragment represents parts of an XML document in the DDOM.
∗ For details on document fragments please check the documentation of the
∗ DocumentFragment interface.
∗
∗ @see org.w3c.dom.DocumentFragment
∗
∗ @author Mathias Neumüller
∗
∗ @version 1.0
∗ /
public class DDocumentFragment extends DGroupNode
    implements Node, DocumentFragment {

    // declaration of constants
    ∗∗ defines the initial size of the structure buffer ∗/
    private static final int INITIAL_SIZE = 1000;

    // declaration of instance variables
    ∗∗ Holds the types of the entries in this document fragment ∗/
    private byte[] type;
    ∗∗ Holds the data indices of the entries in this document fragment ∗/
    private int[] data;
    ∗∗ Holds the number of entries currently held by the document fragment ∗/
    private int size;

    // constructor methods

    ∗∗
    ∗ Default constructor. Creates an empty DocumentFragment with given owner.
    ∗
    ∗ DDocumentFragment(DDocument owner) {
        this(owner, new byte[INITIAL_SIZE], new int[INITIAL_SIZE], 0);
    }

    ∗∗
    ∗ Constructor that creates a DocumentFragment with given content.
    ∗ The content is not copied, i.e. it is a live version of the original.
    ∗
    ∗ DDocumentFragment(DDocument owner, byte[] type, int[] data, int size) {
        super(owner, 0);
        this.type = type;
        this.data = data;
        this.size = size;
        if (size == 0) open();
    }

    ∗∗
    ∗ Constructor that creates a DocumentFragment with given content.
    ∗ The content is not copied, i.e. it is a live version of the original.
    ∗
    ∗ DDocumentFragment(DDocument owner, NodeList children) {
        this(owner);
        appendNodeList(children);
    }
```
C. Source code

```java
// implementation of Node methods

/**
 * Returns the node name of a DocumentFragment, the literal string
 * "#document-fragment"
 */
public String getNodeName() {
    return "#document-fragment";
}

/**
 * Returns the node type of a DocumentFragment, the constant
 * Node.DOCUMENT_FRAGMENT_NODE
 */
public short getType() {
    return Node.DOCUMENT_FRAGMENT_NODE;
}

public Node getParentNode() {
    return null;
}

public Node getPreviousSibling() {
    return null;
}

public Node getNextSibling() {
    return null;
}

public Node appendChild(Node newChild) throws DOMException {
    if (isAppendable()) {
        throw new DOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Can not add more children to this node");
    } else {
        DNode child = (DNode) newChild;
        return child.appendEntry(this, NO_DOMAIN);
    }
}

public Node cloneNode(boolean deep) {
    if (!deep) {
        return new DDocumentFragment(getOwner());
    } else {
        return new DDocumentFragment(getOwner(), getClonedChildNodes());
    }
}

// implementation of DDoucmentFragment methods

/**
 * Returns if the entry with the given index is a type that can have children.
 * I.e., if it is of type document or element.
 */
boolean isParentEntry(int index) {
    return super.isParentEntry(getEntryType(index));
}

/**
 * Returns if the entry with the given index is a type that marks the end of
 * the node.
 */
```
boolean isEndParentEntry(int index) {
    return super.isEndParentEntry(getEntryType(index));
}

/**
 * Returns if the entry with the given index is a type that can be a child of
 * an element. I.e. if it is of type text, element, cdata-section, ...
 */
boolean isChildEntry(int index) {
    return super.isChildEntry(getEntryType(index));
}

void open() {
    appendEntry(getEntryType(), NO_DATA);
    super.open();
}

void close() {
    appendEntry(getEndEntryType(), NO_DATA);
    super.close();
}

/**
 * Appends an entry of given type with given argument to the end of the
 * DocumentFragment. The index of the new entry is returned.
 */
int appendEntry(byte type, int data) {
    if (size >= this.type.length) growStructure();
    this.type[size] = type;
    this.data[size] = data;
    size++;
    return (size-1);
}

/**
 * Doubles the capacity of the data structure.
 */
private void growStructure() {
    byte[] newType = new byte[2*size];
    for (int i=0; i<size; i++) newType[i] = type[i];
    type = newType;
    int[] newData = new int[2*size];
    for (int i=0; i<size; i++) newData[i] = data[i];
    data = newData;
}

/**
 * Returns the entry type of the entry at the given index.
 */
byte getEntryType(int index) {
    return type[index];
}

/**
 * Returns the entry data of the entry at the given index.
 */
int getEntryData(int index) {
    return data[index];
}

/**
 * Returns the index of the parent entry of a entry with specified index.
 */
int getParentEntry(int childindex) {
    int pos = childindex;
    int level = 0;
    if (isParentEntry(pos)) level++;
    while ((level>=0) && (pos >= 0)) {
```java
if (isEndParentEntry(pos)) level++;
if (isParentEntry(pos)) level--;
if (level>=0) pos--;
}
if (level < 0) {
    return pos;
} else {
    return NOENTRY;
}

/**
 * Returns the entry domain of the entry at the given index.
 */
int getEntryDomain(int index) {
    switch (type[index]) {
    case TEXTENTRY:
    case COMMENTENTRY:
    case CDATASECTIONENTRY:
    case ENTITYREFERENCEENTRY:
    case PROCESSINGINSTRUCTIONENTRY:
    case STARTATTRIBUTEENTRY:
    case ENDATTRIBUTEENTRY:
        return getEntryData(getParentEntry(index));
    default:
        return NODOMAIN;
    }
}

/**
 * Returns the entry type used for DocumentFragments.
 */
byte getEntryType() {
    return STARTDOCUMENTFRAGMENTENTRY;
}

/**
 * This method creates a new object referring to the content stored at the
 * given index of this DocumentFragment. If the given entry cannot be
 * represented as DOM object, e.g. an ENDxxxENTRY, this method returns
 * null.
 */
DDocumentFragment getFragment() {
    if (isAppended()) {
        throw new DDOMException(DOMException.NOT_SUPPORTED_ERR,
            "DDocumentFragment error: You should never call on this function once it is appended!" );
    } else {
        return this;
    }
}
```
Node createNode(int index) {
    switch (getEntryType(index)) {
    case STARTDOCUMENTENTRY:
        case STARTDOCUMENTFRAGMENTENTRY:
            //There should only be one DocumentFragment or Document entry in this
            //structure table: The start of this DocumentFragment!
            return this;
        case STARTELEMENTENTRY:
            return new DElement(getFragment(), index);
        case TEXTENTRY:
            return new DText(getFragment(), index);
        case STARTATTRIBUTEENTRY:
            return new DAttr(getFragment(), index);
        case COMMENTENTRY:
            return new DComment(getFragment(), index);
        case CDATASECTIONENTRY:
            return new DCDATASection(getFragment(), index);
        case PROCESSINGINSTRUCTIONENTRY:
            return new DProcessingInstruction(getFragment(), index);
        case ENTITYREFERENCEENTRY:
            throw new java.lang.UnsupportedOperationException(
                "DDocumentFragment.error: EntityReferences cannot be created yet.");
        case STARTDOCUMENTTYPEENTRY:
            throw new java.lang.UnsupportedOperationException(
                "DDocumentFragment.error: DocumentTypes cannot be created yet.");
        default:
            return null;
    }
}

private int[] getEntries(int index, boolean deep) {
    int[] entries;
    if (!isParentEntry(index)) {
        int level = 0;
        int pos = index + 1;
        int entrycounter = 1;
        while (level >= 0) {
            if (deep || (level == 0)) entrycounter++;
            if (isParentEntry(pos)) level++;
            if (isEndParentEntry(pos)) level--;
            pos++;
        }
        entries = new int[entrycounter];
        entrycounter = 0;
        for (pos = index; entrycounter < entries.length; pos++) {
            if (deep || (level <= 0)) entries[entrycounter++] = pos;
            if (isParentEntry(pos)) level++;
            if (isEndParentEntry(pos)) level--;
        }
    } else {
        entries = new int[1];
        entries[0] = index;
    }
    return entries;
}

private int[] getChildren(int index, boolean deep) {
    int[] entries = getEntries(index, deep);
    int childcounter = 0;
    // start at index 1 to avoid returning node as its own child
    for (int i = 1; i < entries.length; i++) {
        if (isChildEntry(entries[i])) childcounter++;
    }
C. Source code

```java
int[] children = new int[childcounter];
childcounter = 0;
for (int i=1; i<entries.length; i++) {
    if (isChildEntry(entries[i])) {
        children[childcounter++] = entries[i];
    }
}
return children;

int[] getAllChildren(int index) {
    return getChildren(index, true);
}

int[] getDirectChildren(int index) {
    return getChildren(index, false);
}

int[] getDirectAttributes(int index) {
    int[] entries = getEntries(index, false);
    int attributecounter = 0;
    for (int i=0; i<entries.length; i++) {
        if (getEntryType(entries[i]) == START_ATTRIBUTE_ENTRY) attributecounter++;
    }
    int[] attributes = new int[attributecounter];
    attributecounter = 0;
    for (int i=0; i<entries.length; i++) {
        if (getEntryType(entries[i]) == START_ATTRIBUTE_ENTRY) {
            attributes[attributecounter++] = entries[i];
        }
    }
    return attributes;
}

int getSize() {
    return size;
}

/**
 * Writes debugging information to the given PrintWriter.
 * The method is supplied for debugging only. It produces a printout of the
 * internal structure array of this DDocumentFragment, followed by the
 * content of the Dictionaries of the owning DDocument.
 * @see Dictionary#dump
 */
public void dump(java.io.PrintWriter writer) {
    writer.println("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -");
    writer.println("Document structure (Entry : Type : Data : Content)");
    writer.println("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -");
    for (int i=0; i<size; i++) {
        writer.println(i+" :"+type[i]+" :"+data[i]+" :"+getDictionaryEntry(type[i], getEntryDomain(i), data[i]).trim());
    }
    writer.println();
    writer.flush();
    getOwner().getDictionary().dump(writer);
}

/**
 * Combines subsequent Text nodes in a DocumentFragment to a single Text node.
 */
```
* The method is used by the Element.normalize() method and exists for compatibility only. Note that the old text entries are not removed from the dictionary, as the Dictionary class currently does not support deletion of entries. A new entry consisting of the concatenation of the two old entries is added to the dictionary, applying this method to a not normalized set of data will increase the memory requirements.

* To avoid this, methods to the Element.normalize() method should be avoided.

* The DParser class makes sure to concatenate subsequent Text entries before they are added to the document structure, thus making calls to this method unnecessary.

* This method should be used with great care only, as it potentially invalidates references to the data structure.

* @see DElement#normalize

```java
void normalize(int domain) {
    if (!isAppended()) {
        // Normalisation of an appended structure would be dangerous.
        // Normalisation may invalidate existing indices!
        boolean changed = false;
        for (int i = 1; i < size; i++) {
            if ((type[i - 1] == TEXTENTRY) && (type[i] == TEXTENTRY)) {
                String text = getDictionaryEntry(TEXTENTRY, domain, data[i - 1]) + getDictionaryEntry(TEXTENTRY, domain, data[i]);
                data[i] = getOwner().getDictionary().append(TEXTENTRY, domain, text);
                type[i - 1] = NOENTRY; // mark entry for deletion
                changed = true;
            }
        }
        if (changed) {
            byte[] newtype = new byte[type.length];
            int[] newdata = new int[data.length];
            int newsize = 0;
            for (int i = 0; i < size; i++) {
                // delete entries that are not used any more
                if (type[i] != NOENTRY) {
                    newtype[newsize] = type[i];
                    newdata[newsize] = data[i];
                    newsize++;
                }
            }
            type = newtype;
            data = newdata;
            size = newsize;
        }
    }
}
```

// implementation of DGroupNode methods

```java
byte getEndEntryType() {
    return DNode.ENDDOCUMENTFRAGMENTENTRY;
}
```

**Class ddom.dom.DDocument**

```java
package ddom.dom;
```
import org.w3c.dom.*;
import java.util.List;

/**
 * The class DDocument represents an entire XML document in the DOM
 * implementation. All nodes created using this document are stored using
 * centrally managed dictionaries in order to save memory. For details on
 * documents please check the documentation of the Document interface.
 *
 * @see org.w3c.dom.Document
 * @author Mathias Neumüller
 * @version 1.0
 */
public class DDocument extends DDocumentFragment implements Node, DocumentFragment, Document {

    // declaration of instance variables
    Dictionary dict;
    DDocumentType doctype;

    // constructor methods

    /**
     * Constructor that creates a new, empty document. This method is
     * not public and should only be used by the cloneNode() method.
     */
    DDocument(DDocumentType doctype, NodeList children) {
    }

    public DDocument(DDocumentType doctype) {
        super(null);
        dict = new Dictionary();
        this.doctype = doctype;
    }

    public DDocument() {
        this(null);
    }

    /**
     * Standard constructor used to create a new, empty document. 
     * Every document created using the DOM package will have its own
     * dictionary. Once a document is created using this constructor, child nodes
     * can be created and appended using the createXXX() and appendChild() 
     * methods. 
     * A more generic way to create new documents is to use the Java
     * API for XML Processing (JAXP). DOM is JAXP compliant and new documents can 
     * be created by setting the system property <CODE>
     * "javax.xml.parsers.DocumentBuilderFactory"<CODE> to <CODE>
     * "ddom.jaxp.DDocumentBuilderFactory"<CODE> and using the appropriate factory
     * methods. 
     * System.setProperty( "javax.xml.parsers.DocumentBuilderFactory", 
     * "ddom.jaxp.DDocumentBuilderFactory" );<BR>
     * DocumentBuilderFactory factory = 
     * DocumentBuilderFactory.newInstance();<BR>
     * DocumentBuilder builder = factory.newDocumentBuilder();<BR>
     * Document document = builder.newDocument();<BR>
     * </CODE>
     * @see ddom.jaxp.DDocumentBuilderFactory
     * @see ddom.jaxp.DDocumentBuilder
     */
    public DDocument() {
    }
}
C. Source code

```java
this(doctype);
appendNodeList(children);
}

// implementation of Document methods

public DocumentType getDoctype() {
    if (isAppended()) {
        int index = 0;
        boolean found = false;
        while (!found && (index < getSize()) &&
            (getEntryType(index) != ENDDOCUMENTENTRY)) {
            index = (getEntryType(index) == STARTDOCUMENTENTRY);
            if (found) {
                return (DocumentType) createNode(index - 1);
            } else {
                return null;
            }
        }
    } else {
        return doctype;
    }
}

public DOMImplementation getImplementation() {
    return new DDOMImplementation();
}

public Element getDocumentElement() {
    int index = 0;
    boolean found = false;
    while (!found && (index < getSize()) &&
        (getEntryType(index) != ENDOBJECTENTRY)) {
        index = (getEntryType(index) == STARTOBJECTENTRY);
        if (found) {
            return (Element) createNode(index - 1);
        } else {
            return null;
        }
    }
}

public Element createElement(String tagName) throws DOMException {
    return new DElement(getOwner(), tagName);
}

public DocumentFragment createDocumentFragment() {
    return new DDocumentFragment(getOwner());
}

public Text createTextNode(String data) {
    return new DText(getOwner(), data);
}

public Comment createComment(String data) {
    return new DComment(getOwner(), data);
}

public CDATASection createCDATASection(String data) throws DOMException {
    return new DC DATASection(getOwner(), data);
}
```
C. Source code

```java
public ProcessingInstruction createProcessingInstruction(String target, String data) throws DOMException {
    return new DProcessingInstruction(getOwner(), target, data);
}

public Attr createAttribute(String name) throws DOMException {
    return new DAttr(getOwner(), name, true);
}

public EntityReference createEntityReference(String name) throws DOMException {
    return new DEntityReference(getOwner(), name);
}

public NodeList getElementsByTagName(String tagname) {
    return getDocumentElement().getElementsByTagName(tagname);
}

// implementations of Node methods

public String getNodeName() {
    return "#document";
}

public short getNodeType() {
    return Node.DOCUMENT_NODE;
}

public Node cloneNode(boolean deep) {
    if (!deep) {
        return new DDocument((DDocumentType) getDoctype());
    } else {
        return new DDocument((DDocumentType) getDoctype(), getClonedChildNodes());
    }
}

// implementation of DDocument methods

/**
 * Returns the dictionary used by this document.
 */
Dictionary getDictionary() {
    return dict;
}

/**
 * Creates a new DocumentType with the given name and sets it to be the
 * DocumentType of this Document. This method can only be called by called
 * once and is not part of the DOM level 1 interface. It should only be called
 * by parsers.
 *
 * @exception DOMException if DocumentType was already set.
 */
void setDocumentType(String name) throws DOMException {
    if (isAppended() || (doctype != null)) {
        throw new DDOMException(DOMException.MODIFICATION_ALLOWED_ERR,
            "DDocument_error::Document_type_can_only_be_set_once!");
    } else {
        doctype = new DDocumentType(getOwner(), name);
    }
}
```
// implementation of DGroupNode methods

byte getEndEntryType() {
    return DNode.ENDDOCUMENTENTRY;
}

// implementation of DNode methods

byte getEntryType() {
    return STARTDOCUMENTENTRY;
}

DDocument getOwner() {
    return this;
}

DDocumentFragment getFragment() {
    return this;
}

}

Class ddom.dom.DDocumentType

package ddom.dom;

import org.w3c.dom.*;

/**
 * The class DDocumentType represents the type of a DDOM document.
 * It implements the DocumentType interface. For details about this document
 * types please refer to the documentation of this interface.</P>
 * 
 * Note: The current version of the DDOM implementation does not handle
 * document type information correctly. This class is rather a spaceholder for
 * future developments.</P>
 * 
 * @see org.w3c.dom.DocumentType
 * 
 * @author Mathias Neumüller
 * 
 * @version 1.0
 */

public class DDocumentType extends DDocumentFragment implements DocumentType {

    // Proper handling of document types is not implemented yet. Only the name
    // of the document class can be stored using this class.
    // Entities and Notations, which are stored as properties of the document type
    // are not yet implemented.
    // However, this class is implemented as a child class of the DDocument class
    // to allow easy management of children in the future.

    DDocumentType(DDocument owner, String name) {
        super(owner);
        setEntryIndex(appendDictionaryEntry(getEntryType(), name));
    }

    public String getName() {
        return getDictionaryEntry(getEntryType(), getEntryIndex());
    }

    public NamedNodeMap getEntities() {

/** @todo: Implement this org.w3c.dom.DocumentType methods*/
throw new DDOMException(DOMException.NOT_SUPPORTED_ERR,
    "DDocumentType.error: Entities are currently not supported in the DOM");
}

public NamedNodeMap getNotations() {
/** @todo: Implement this org.w3c.dom.DocumentType methods*/
throw new DDOMException(DOMException.NOT_SUPPORTED_ERR,
    "DDocumentType.error: Notations are currently not supported in the DOM");
}

public String getNodeName() {
    return getName();
}

public void setNodeValue(String nodeValue) throws DOMException {
    throw new DDOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
        "DDocumentType.error: DocumentType nodes are readonly nodes in " +
        "DOM level 1.");
}

public short getNodeType() {
    return Node.DOCUMENTTYPE_NODE;
}

// implementation of DNode methods
byte getEntryType() {
    return DNode.START_DOCUMENTTYPEENTRY;
}

// implementation of DGroupNode methods
byte getEndEntryType() {
    return DNode.END_DOCUMENTTYPEENTRY;
}

Class ddom.dom.DElement

package ddom.dom;

import org.w3c.dom.*;

/**
 * <P>The class DElement represents an element in the DOM. It implements the
 * Element interface. For details on elements check the documentation
 * of the Element interface.</P>
 *
 * <P>Note: There is no public constructor available for this class. You can get
 * instances of this class by calling the <CODE>Document.createElement()</CODE>
 * method, or by using the <CODE>cloneNode()</CODE> method.</P>
 *
 * @see org.w3c.dom.Element
 * @see org.w3c.dom.Document#createElement
 * @author Mathias Neumuller
 * @version 1.0
 */

public class DElement extends DGroupNode implements Node, Element{
C. Source code

// declaration of instance variables

// used to buffer children if the element itself has not been appended
private DDocumentFragment buffer;

// used to buffer attributes if the element itself has not been appended
private DNamedNodeMap attributes;

// constructor methods

/**
 * Standard constructor, can be used to create DElement objects referring to
 * elements already appended to a parent node.
 */
DElement(DDocumentFragment owner, int index) {
    super(owner, index);
    buffer = null;
    attributes = null;
}

/**
 * Constructor to create a new DElement with the given name without a parent
 * node.
 */
DElement(DDocument owner, String tagName) {
    this(owner, NOT_IN_TREE);
    setEntryIndex(appendDictionaryEntry(getEntryType(), tagName));
}

/**
 * Constructor to create an new DElement with the given name and given
 * attributes but without children as used by the cloneNode(false) method.
 */
private DElement(DDocument owner, String tagName, DNamedNodeMap attr) {
    this(owner, tagName);
    attributes = attr;
}

/**
 * Constructor to create an new DElement with the given name and given
 * attributes and children as used by the cloneNode(true) method.
 */
private DElement(DDocument owner, String tagName, DNamedNodeMap attr,
                 NodeList children) {
    this(owner, tagName, attr);
    appendNodeList(children);
}

// implementation of Element methods

public String getTag() {
    return getNodeName();
}

public String getAttribute(String name) {
    return getAttributeNode(name).getValue();
}

public void setAttribute(String name, String value) throws DOMException {
    Attr attribute = getOwner().createAttribute(name);
    attribute.setValue(value);
    setAttributeNode(attribute);
}
public void removeAttribute(String name) throws DOMException {
    if (isAppended()) {
        throw new DDOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Modification of element entries after appending are currently not allowed.");
    } else {
        attributes.removeNamedItem(name);
    }
}

public Attr getAttributeNode(String name) {
    return (Attr) getAttributes().getNamedItem(name);
}

public Attr setAttributeNode(Attr newAttr) throws DOMException {
    if (isAppended()) {
        throw new DDOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Can not add further attributes to this node");
    } else {
        if (attributes == null) attributes = new DNamedNodeMap();
        return (Attr) attributes.setNamedItem(newAttr);
    }
}

public Attr removeAttributeNode(Attr oldAttr) throws DOMException {
    if (isAppended()) {
        throw new DDOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "DElement error: Can not remove attributes from an already appended element.");
    } else {
        if (attributes != null) {
            Node oldnode = attributes.removeNamedItem(oldAttr.getName());
            if (oldnode == null) {
                throw new DDOMException(DOMException.NOT_FOUND_ERR,
                    "Attribute not found error.");
            } else {
                return (Attr) oldnode;
            }
        } else {
            throw new DDOMException(DOMException.NOT_FOUND_ERR,
                "Attribute not found error.");
        }
    }
}

public NodeList getElementsByTagName(String name) {
    DNodeList nl = new DNodeList();
    DDocumentFragment frag = getFragment();
    int tag;
    if (name.equals("*")) {
        tag = DNode.ALL_DATA;
    } else {
        tag = getOwner().getDictionary().find(getEntryType(),
            getEntryDomain(), name);
        // if there is no entry for this element name, there won’t be a node!
        if (tag == NO_DATA) return nl;
    }
    int[] childrenref = frag.getAllChildren(getEntryIndex());
    for (int i = 0; i < childrenref.length; i++) {
        if (((frag.getEntryType(childrenref[i]) == DNode.START_ELEMENT_ENTRY) &&
            (tag == DNode.ALL_DATA) ||
            ((frag.getEntryType(childrenref[i]) == DNode.END_ELEMENT_ENTRY) &
            (tag == DNode.ALL_DATA) ||
            ((frag.getEntryType(childrenref[i]) == DNode.TEXTENTRY) &
                (tag == DNode.ALL_DATA))) ||
            ((frag.getEntryType(childrenref[i]) == DNode.COMMENTENTRY) &
                (tag == DNode.ALL_DATA) ||
            ((frag.getEntryType(childrenref[i]) == DNode.ENTITYREFERENCEENTRY) &
                (tag == DNode.ALL_DATA) ||
            ((frag.getEntryType(childrenref[i]) == DNode.ENTITYDEFENTRY) &
                (tag == DNode.ALL_DATA)) ||
            ((frag.getEntryType(childrenref[i]) == DNode.REFERENCEENTRY) &
                (tag == DNode.ALL_DATA) ||
            ((frag.getEntryType(childrenref[i]) == DNode.ENTITYREFENTRY) &
                (tag == DNode.ALL_DATA) ||
            ((frag.getEntryType(childrenref[i]) == DNode.PROCESSINGINDEXTENTRY) &
                (tag == DNode.ALL_Data) ||
            ((frag.getEntryType(childrenref[i]) == DNode.PROCESSINGINDEXTENTRY) &
                (tag == DNode.ALL_DATA)))))) {
            nl.appendChild(frag.getNodeAt(childrenref[i]));
        }
    }
    return nl;
}
C. Source code

```java
(tag == frag.getEntryData(childrenref[i]))) {
    nl.append(frag.createNode(childrenref[i]));
}
return nl;
}

public void normalize() {
    if (isAppended()) {
        throw new DOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "DElement_error: Cannot normalise an already appended element.");
    } else {
        if (buffer != null) buffer.normalize(getDataIndex());
    }
}

// implementation of Node methods

public String getNodeName() {
    return getDictionaryEntry(getEntryType(), getEntryDomain(), getDataIndex());
}

public short getNodeType() {
    return Node.ELEMENT_NODE;
}

public NodeList getChildNodes() {
    if (isAppended()) {
        return super.getChildNodes();
    } else {
        if (buffer == null) {
            return new DNodeList();
        } else {
            return buffer.getChildNodes();
        }
    }
}

public NamedNodeMap getAttributes() {
    DNamedNodeMap attr;
    if (isAppended()) {
        attr = new DNamedNodeMap();
        int[] atts = getFragment().getDirectAttributes(getEntryIndex());
        for (int i = 0; i < atts.length; i++) {
            attr.setNamedItem(getFragment().createNode(atts[i]));
        }
    } else {
        return attr = attributes;
    }
    return attr;
}

public Node appendChild(Node newChild) throws DOMException {
    if (isAppended()) {
        throw new DOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Can not add further children to this node");
    } else {
        if (buffer == null) buffer = new DDocumentFragment(getOwner());
        DNode child = (DNode) newChild;
        return child.appendEntry(buffer, getEntryIndex());
    }
}
```
public boolean hasChildNodes() {
    if (isAppended()) {
        return super.hasChildNodes();
    } else {
        if (buffer == null) {
            return false;
        } else {
            return buffer.hasChildNodes();
        }
    }
}

public Node cloneNode(boolean deep) {
    if (!deep) {
        return new DElement(getOwner(), getTagName(), (DNamedNodeMap)getAttribute());
    } else {
        return new DElement(getOwner(), getTagName(), (DNamedNodeMap)getAttribute(), getClonedChildNodes());
    }
}

// implementation of internal methods

byte getEntryType() {
    return START_ELEMENT ENTRY;
}

DNode appendEntry(DDocumentFragment doc, int domain) {
    if (!isAppended()) {
        int index = doc.appendEntry(getEntryType(), getEntryIndex());
        if (attributes != null) {
            for (int i = 0; i < attributes.getLength(); i++){
                ((DAttr)attributes.item(i)).appendEntry(doc, getEntryIndex());
            }
            attributes = null;
        }
        if (buffer != null) {
            buffer.close();
            buffer.appendEntry(doc, getEntryIndex());
            buffer = null;
        }
    doc.appendEntry(getEndEntryType(), getEntryIndex());
    super.appendEntry(doc, domain);
    return new DElement(getOwner(), index);
} else {
    throw new DDOMException(DOMException.NOT_SUPPORTED_ERR,
        "DElement error: This node was already appended somewhere, it can not be appended again."");
}

DDocumentFragment getFragment() {
    if (isAppended()) {
        return super.getFragment();
    } else {
        return buffer;
    }
}

// implementation of DGroupNode methods
```java
byte getEndEntryType() {
    return DNode.END_ELEMENT_ENTRY;
}
}

Class ddom.dom.DCharacterData
package ddom.dom;

import org.w3c.dom.*;

/**
 * The abstract class DCharacterData is the base for all classes representing
 * textual data. It implements the CharacterData interface and is the super
 * class of the classes DText, DComment and DCDATASection. Most of the text
 * specific functionality is implemented here. For more information about
 * character data please read the documentation of the CharacterData interface.
 * @see org.w3c.dom.CharacterData
 * @author Mathias Neumüller
 * @version 1.0
 */
abstract public class DCharacterData extends DNode implements Node, CharacterData {
    // declaration of instance variables

    /**
     * Used to store the content of the node before it is appended.
     * For an appended node this is always null and the value is determined
     * using the central dictionary.
     */
    String text;

    // constructor methods

    /**
     * Constructor used to create a reference to a textual node with a DOM tree.
     */
    DCharacterData(DDocumentFragment owner, int index) {
        super(owner, index);
        text = null;
    }

    /**
     * Constructor used to create a new textual node containing the given text.
     */
    DCharacterData(DDocument owner, String text) {
        this(owner, NOT_IN_TREE);
        // make a deep copy here to avoid external manipulation of the content.
        this.text = text.toString();
    }

    // implementation of CharacterData methods

    public String getData() throws DOMException {
        if (isAppended()) {
            return getDictionaryEntry(getEntryType(), getEntryDomain());
        }
    }
```
C. Source code

```
getDataIndex();

} else {
    return text;
}

/**
 * Sets the data of this node to the given value.
 * @exception DOMException NO_MODIFICATION_ALLOWED_ERR
 * if the node is readonly, i.e. if it was already appended.
 */
public void setData(String data) throws DOMException {
    if (isAppended()) {
        throw new DDOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Modification of text entries after appending are currently not allowed.");
    } else {
        text = data;
    }
}

public int getLength() {
    String s = getData();
    if (s != null) {
        return s.length();
    } else {
        return 0;
    }
}

public String substringData(int offset, int count) throws DOMException {
    return getData().substring(offset, count);
}

public void appendData(String arg) throws DOMException {
    if (isAppended()) {
        throw new DDOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Modification of text entries after appending are currently not allowed.");
    } else {
        if (text == null) {
            text = arg;
        } else {
            if (arg != null) text = text+arg;
        }
    }
}

public void insertData(int offset, String arg) throws DOMException {
    if (isAppended()) {
        throw new DDOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Modification of text entries after appending are currently not allowed.");
    } else {
        if ((text == null) || (offset < 0) || (offset > text.length())) {
            throw new DDOMException(DOMException.INDEX_SIZE_ERR,
                "DCharacterData_error::Invalid_index_for_insertion_of_data.");
        } else {
            if (arg != null) text = text.substring(0, offset)+arg+
                text.substring(offset);
        }
    }
```


public void deleteData(int offset, int count) throws DOMException {
    if (isAppended()) {
        throw new DOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
                "Modification of text entries after appending are currently not allowed.");
    } else {
        if ((text == null) || (offset < 0) || (offset > text.length()) ||
                (count < 0)) {
            throw new DOMException(DOMException.INVALID_MODIFICATION_ERR,
                    "Invalid modification of data.");
        } else {
            if (text.length() < (offset+count)) {
                text = text.substring(0, offset);
            } else {
                text = text.substring(0, offset)+text.substring(offset+count);
            }
        }
    }
}

public void replaceData(int offset, int count, String arg)
    throws DOMException {
    if (isAppended()) {
        throw new DOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
                "Modification of text entries after appending are currently not allowed.");
    } else {
        deleteData(offset, count);
        insertData(offset, arg);
    }
}

// implementation of Node methods

public String getNodeValue() throws DOMException {
    return getData();
}

public void setNodeValue(String nodeValue) throws DOMException {
    setData(nodeValue);
}

// implementation of DNode methods

/*
 * Appends this entry to the given DocumentFragment within the given domain.
 */
DNode appendEntry(DDocumentFragment doc, int domain) throws DOMException {
    checkValidy(doc);
    setEntryIndex(appendDictionaryEntry(getEntryType(), domain, text));
    text = null;
    DNode n = (DNode) doc.createNode(doc.appendEntry(getEntryType(),
                getDataIndex()));
}
```java
    close();
    return n;
}
}

Class ddom.dom.DComment

package ddom.dom;

import org.w3c.dom.*;

/**
 * The class DComment represents comments in the DDOM tree. It implements the
 * Comment interface. For more information about comments please refer to this
 * interface.</P>
 * Note: There is no public constructor available for this class. You can get
 * instances of this class by calling the Document.createComment() method, or by using the cloneNode() method.<P>
 * @see org.w3c.dom.Comment
 * @see org.w3c.dom.Document#createComment
 * @author Mathias Neumüller
 * @version 1.0
 */

public class DComment extends DCharacterData implements Node, CharacterData, Comment {
    // constructor methods

    /**
     * Constructor used to create a reference to a comment node at the specified
     * position.
     */
    DComment(DDocumentFragment owner, int index) {
        super(owner, index);
    }

    /**
     * Constructor used to create a new comment node with the given content.
     */
    DComment(DDocument owner, String text) {
        super(owner, text);
    }

    // implementation of Node methods

    /**
     * Returns the name of this node, the literal String "#comment".
     */
    public String getNodeName() {
        return "#comment";
    }

    /**
     * Returns the type of this node, the constant Node.COMMENT_NODE.
     */
    public short getType() {
        return Node.COMMENT_NODE;
    }

    public Node cloneNode(boolean deep) {
        return new DComment(getOwner(), getData());
    }

```
// implementation of DNode methods

byte getEntryType() {
    return DNode.COMMENTENTRY;
}

Class ddom.dom.DText

package ddom.dom;

import org.w3c.dom.∗;

/** *
 * The class DText represents text nodes on the DOM implementation. It *
 * implements the Text interface which is used to represent parsed text. *
 * For further information please refer to the documentation of the Text *
 * interface.</P> *
 *
 * Note: There is no public constructor available for this class. You can get *
 * instances of this class by calling the <CODE>Document.createTextNode()</CODE> *
 * method or by using the <CODE>cloneNode()</CODE> method.</P> *
 *
 * @see org.w3c.dom.Text
 * @see org.w3c.dom.Document#createTextNode
 * @author Mathias Neumüller
 * @version 1.0
 */

public class DText extends DCharacterData implements Node, CharacterData, Text {

    // constructor methods

    /** *
     * Constructor used to create a reference to a text node at the specified *
     * position.
     */
    DText(DDocumentFragment owner, int index) {
        super(owner, index);
    }

    /** *
     * Constructor used to create a new text node with the given content.
     */
    DText(DDocument owner, String text) {
        super(owner, text);
    }

    // implementation of Text methods

    /** *
     * Splits this text node into two at the given index.<P>
     * Note: This method is currently not supported by the DOM implementation.
     * Every call on this method will result in a DOMException being thrown.<P>
     * @exception DOMException NOT_SUPPORTED_ERR always.
     */
    public Text splitText(int offset) throws DOMException {
        throw new DOMException(DOMException.NOT_SUPPORTED_ERR, "DText error: SPLITTING:text_nodes_is_currently_not_supported_by_the_DDOM.");
    }

    // implementation of Node methods
C. Source code

```java
/** Returns the name of this node, the literal String "#text". */
public String getNodeName() {
    return "#text";
}

/** Returns the type of this node, the constant Node.TEXT_NODE. */
public short getNodeType() {
    return Node.TEXT_NODE;
}

public Node cloneNode(boolean deep) {
    return new DText(getOwner(), getData());
}

// implementation of DNode methods
byte getEntryType() {
    return TEXTENTRY;
}
```

Class ddom.dom.DCDATASection

```java
package ddom.dom;

import org.w3c.dom.*;

/**
 * The class DCDATASection is used to represent CDATA sections. That is
 * textual data that is not to be parsed. It implements the interface
 * CDATASection. To learn more about CDATA sections, please refer to the
 * documentation of the CDATASection interface.</P>
 * <P>Note: There is no public constructor available for this class. You can get
 * instances of this class by calling the <CODE>Document.createCDATASection()</CODE>
 * method, or by using the <CODE>cloneNode()</CODE> method.</P>
 * @see org.w3c.dom.CDATASection
 * @see org.w3c.dom.Document#createCDATASection
 * @author Mathias Neumüller
 * @version 1.0
 */

public class DCDATASection extends DText
    implements Node, CharacterData, Text, CDATASection {

    // constructor methods

    /**
     * Constructor used to create a reference to a cdata–section node at the
     * specified position.
     */
    DCDATASection(DDocumentFragment owner, int index) {
        super(owner, index);
    }

    /**
     * Constructor used to create a new cdata–section node with the given content.
     */
```
C. Source code

DCDATASection(DDocument owner, String text) {
    super(owner, text);
}

// implementation of Node methods

/**
 * Returns the name of this node, the literal String "#cdata-section".
 */
public String getNodeName() {
    return "#cdata-section";
}

/**
 * Returns the type of this node, the constant Node.CDATASECTION_NODE.
 */
public short getNodeType() {
    return Node.CDATASECTION_NODE;
}

public Node cloneNode(boolean deep) {
    return new DCDATASection(getOwner(), getData());
}

// implementation of DNode methods

byte getEntryType() {
    return DNode.CDATASECTIONENTRY;
}

Class ddom.dom.DAttr

package ddom.dom;

import org.w3c.dom.Attr;
import org.w3c.dom.Node;
import org.w3c.dom NodeList;
import org.w3c.dom NamedNodeMap;
import org.w3c.dom Document;
import org.w3c.dom.DOMException;

/**
 * The class DAttr is used to represent attribute nodes in the DOM. It is
 * implementing the Attr interface. For details on attributes check the
 * documentation of the Attr interface.</P>
 *
 * Note: There is no public constructor available for this class. You can get
 * instances of this class by calling the <CODE>Document.createAttribute()</CODE> method or by using the <CODE>Element.setAttribute()</CODE> or the
 * <CODE>Attr.cloneNode()</CODE> method.</P>
 *
 * @see org.w3c.dom.Attr
 * @see org.w3c.dom.Document#createAttribute
 * @see org.w3c.dom.Element#setAttribute
 * @see #cloneNode
 * @see org.w3c.dom.Mathias Neumann;11er
 * @version 1.0
 */
public class DAttr extends DGroupNode implements Node, Attr {
    // declaration of instance variables

    // used to buffer appended children of this attribute
C. Source code

```java
private DNodeList buffer;
// the name of this attribute as long as its domain is unknown
String name;
// this does not work properly yet, just included for future implementation
boolean specified;

// constructor methods
DAAttr(DDocumentFragment owner, int index) {
    super(owner, index);
    name = null;
    buffer = null;
    specified = false; // need to store these properties somehow in the structure
}

DAAttr(DDocument owner, String name, boolean specified) {
    this(owner, NOT_IN_TREE);
    this.name = name;
    this.specified = specified;
}

DAAttr(DDocument owner, String name, boolean specified, DNodeList children) {
    this(owner, name, specified);
    buffer = children;
}

DAAttr(DDocument owner, String name, boolean specified, String value) {
    this(owner, name, specified);
    buffer = new DNodeList();
    DText child = new DText(getOwner(), value);
    buffer.append(child);
}

// implementation of Attr methods
public String getName() {
    if (isAppended()) {
        return getDictionaryEntry(getEntryType(), getEntryDomain(),
                                     getDataIndex());
    } else {
        return name;
    }
}

/**
 * If this attribute was explicitly given a value in the original document,
 * this is <CODE>true</CODE>; otherwise, it is <CODE>false</CODE>.
 * Note that this is not handled properly by the DOM implementation yet.
 * This method will return <CODE>true</CODE> as long as the methods
 * <CODE>setValue</CODE> or <CODE>setNodeValue</CODE> have not been called and
 * <CODE>false</CODE> otherwise.
 */
public boolean getSpecified() {
    /*
     * @todo: This needs rethinking: How to preserve this state once the
     * attribute is appended to the structure? Use additional entry or
     * different entry types for specified and unspecified entries?
     */
    return specified;
}

public String getValue() {
```
```
C. Source code

```java
StringBuffer sb = new StringBuffer();
if (isAppended()) {
    int [] children = getFragment().getDirectChildren(getEntryIndex());
    for (int i = 0; i < children.length; i++) {
        sb.append(getFragment().createNode(children[i]).getNodeValue());
    }
    return sb.toString();
} else {
    if (buffer != null) {
        for (int i = 0; i < buffer.getLength(); i++) {
            sb.append(buffer.item(i).getNodeValue());
        }
        return sb.toString();
    } else {
        return null;
    }
}

/** *
 * Sets the value of the attribute to the given String. That is a new text *
 * node will be created containing the given text which will be attached to *
 * the attribute as only child.
 */
public void setValue(String value) {
    if (isAppended()) {
        throw new DOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Can_not_change_the_value_of_this_attribute.");
    } else {
        buffer = new DNodeList(); // clear previous content
        DText child = new DText(getOwner(), value);
        buffer.append(child);
        specified = true;
    }
}

// implementation of Node methods

/** *
 * Returns the name of the attribute.
 * @see #getName
 */
public String getNodeName() {
    return getName();
}

/** *
 * Returns the value of the attribute.
 * @see #getValue
 */
public String getNodeValue() throws DOMException {
    return getValue();
}

/** *
 * Sets the value of the attribute.
 * @see #setValue
 */
public void setNodeValue(String nodeValue) throws DOMException {
    setValue(nodeValue);
}
```
C. Source code

```java
/** Returns the type of the node, the constant Node.ATTRIBUTE_NODE. */
public short getNodeType() {
    return Node.ATTRIBUTE_NODE;
}

/**
 * Returns the parent of this node. Note that attributes are not part of the
 * DOM tree but properties of their containing elements. Hence this method
 * will always return <code>null</code>.
 */
public Node getParentNode() {
    return null;
}

public NodeList getChildNodes() {
    if (isAppended()) {
        return super.getChildNodes();
    } else {
        return buffer;
    }
}

public Node appendChild(Node newChild) throws DOMException {
    if (isAppended()) {
        throw new DOMException(DOMException.NO_MODIFICATION_ALLOWED_ERR,
            "Can not add further children to this node");
    } else {
        if (buffer == null) buffer = new DNodeList();
        buffer.appendChild(newChild);
        return newChild;
    }
}

public Node cloneNode(boolean deep) {
    return new DAttr(getOwner(), getName(), getSpecified(),
        (DNodeList) getClonedChildNodes());
}

// implementation of DNode methods
byte getEntryType() {
    return DNode.START_ATTRIBUTE_ENTRY;
}

DNode appendEntry(DDocumentFragment doc, int domain) {
    setEntryIndex(appendDictionaryEntry(getEntryType(), domain, name));
    name = null;
    int index = doc.appendChild(getEntryType(), getDataIndex());
    if (buffer != null) {
        for (int i=0; i<buffer.getLength(); i++) {
            ((DNode) buffer.item(i)).appendEntry(doc, domain);
        }
    }
    doc.appendChild(getEndEntryType(), getDataIndex());
    super.appendChild(doc, domain);
    return (DNode) doc.createNode(index);
}

// implementation of DGroupNode methods
byte getEndEntryType() {
```
Class ddom.dom.DEntityReference

package ddom.dom;

import org.w3c.dom.∗;

/**
 * The class DEntityReference represents an entity reference node in the DOM tree. For details on EntityReferences check the documentation of the EntityReference interface. </P>
 * Note: Parsers usually resolve entities while they are parsing. Hence this node will not appear in a DOM tree under normal circumstances. Nevertheless a parser may choose not to resolve entities under certain condition in which case nodes of this type will occur in the tree. Most of the functionality needed to represent entities properly is currently not implemented by the DDOM architecture. </P>
 * @see org.w3c.dom.EntityReference
 * @author Mathias Neumüller
 * @version 1.0
 */
public class DEntityReference extends DCharacterData implements Node, EntityReference {

    // constructor methods

    /**
     * Constructor used to create a reference to an entity reference node at the specified position.
     */
    DEntityReference(DDocumentFragment owner, int index) {
        super(owner, index);
    }

    /**
     * Constructor used to create a new entity reference node with the given content.
     */
    DEntityReference(DDocument owner, String text) {
        super(owner, text);
    }

    // implementation of Node methods

    public short getNodeType() {
        return Node.ENTITYREFERENCE_NODE;
    }

    public String getNodeName() {
        return super.getData();
    }

    public String getNodeValue() {
        return null;
    }
}
public void setNodeValue(String s) {
    throw new DOMException(DOMException.NO_DATA_ALLOWED_ERR,
        "DEntityReference.error: You cannot set the node.value of this type!" );
}

public Node cloneNode(boolean deep) {
    return new DEntityReference(getOwner(), getData());
}

// implementation of DNode methods
byte getEntryType() {
    return DNode.ENTITY_REFERENCE_ENTRY;
}

Class ddom.dom.DProcessingInstruction

package ddom.dom;

import org.w3c.dom.*;

/**
 * The class DProcessingInstruction represents an processing instruction (PI)
 * in the DOM tree. For details on PIs check the documentation of the
 * ProcessingInstruction interface.</P>
 * @see org.w3c.dom.ProcessingInstruction
 * @author Mathias Neumüller
 * @version 1.0
 */
public class DProcessingInstruction extends DCharacterData
    implements ProcessingInstruction {
    // constructor methods
    DProcessingInstruction(DDocument owner, String target, String data) {
        super(owner, target+SEPARATOR+data);
    }
    DProcessingInstruction(DDocument owner, String text) {
        super(owner, text);
    }
    DProcessingInstruction(DDocumentFragment owner, int index) {
        super(owner, index);
    }

    // implementation of ProcessingInstruction methods
    public String getTarget() {
        String pi = super.getData();
        if (pi == null) {
            return null;
        } else {
            return pi.substring(0, pi.indexOf(SEPARATOR));
        }
    }
public String getData() {
    String pi = super.getData();
    if (pi == null) {
        return null;
    } else {
        return pi.substring(pi.indexOf(SEPARATOR)+1);
    }
}

public void setData(String data) throws DOMException {
    String newdata = getTarget()+SEPARATOR+data;
    super.setData(newdata);
}

public String getNodeName() {
    return getTarget();
}

public String getNodeValue() throws DOMException {
    return getData();
}

public void setNodeValue(String nodeValue) throws DOMException {
    setData(nodeValue);
}

public short getNodeType() {
    return Node.PROCESSING_INSTRUCTION_NODE;
}

public Node cloneNode(boolean deep) {
    return new DText(getOwner(), super.getData());
}

class ddom.dom.DNodeList
}

package ddom.dom;

import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import java.util.ArrayList;
import java.util.List;

/**
 * The class DNodeList represents a list of nodes. This class implements the
 * NodeList interface and is used to handle sets of nodes as returned by the
 * Node.getChildNodes() method. For further information on this class please
 * check the NodeList interface.
 * @see org.w3c.dom.NodeList
 */
public class DNodeList implements NodeList {

    // declare instance variables

    /** Internal List variable used to store the nodes of this list. */
    List nodes;

    // constructor methods

    /** Default constructor used to create an empty DNodeList. */
    public DNodeList() {
        nodes = new ArrayList();
    }

    // implementation of NodeList methods

    public Node item(int index) {
        if ((index < 0) || (index >= nodes.size())) {
            return null;
        } else {
            return (Node) nodes.get(index);
        }
    }

    public int getLength() {
        return nodes.size();
    }

    // implementation of DNodeList methods

    /** Appends a new node to the end of this node list. */
    protected void append(Node n) {
        nodes.add(n);
    }

    /** Appends the nodes hold by another nodelist to the end of this node list. */
    protected void append(NodeList n) {
        if (n != null) {
            for (int i=0; i<n.getLength(); i++) {
                nodes.add(n.item(i));
            }
        }
    }

}
The class DNamedNodeMap represents a list of nodes that can be addressed by their name. This class implements the NamedNodeMap interface and is used to handle sets of attributes as returned by the Node.getAttributes() method. For further information on this class please check the NamedNodeMap interface.

@see org.w3c.dom.NamedNodeMap

@version 1.0

public class DNamedNodeMap implements NamedNodeMap {

    // declaration of instance variables
    @private Map map;

    public DNamedNodeMap() {
        map = new HashMap();
    }

    public Node getNamedItem(String name) {
        return (Node) map.get(name);
    }

    public Node setNamedItem(Node arg) throws DOMException {
        return (Node) map.put(arg.getNodeName(), arg);
    }

    public Node removeNamedItem(String name) throws DOMException {
        return (Node) map.remove(name);
    }

    public Node item(int index) {
        return (Node) (map.values().toArray()[index]);
    }

    public int getLength() {
        return map.size();
    }

}

Class ddom.dom.DDOMException

package ddom.dom;

import org.w3c.dom.DOMException;

/**
 * The class DDOMException implements the DOMException interface.
 * DOMExceptions are raised in case an operation would violate the XML standard
 * or is impossible to perform for technical reasons. For details about
 * DOMException please refer to the DOMException interface.
 */

@see org.w3c.dom.DOMException
C. Source code

```java
@Author Mathias Neum&ouml;ller
@Version 1.0 */

public class DDOMException extends DOMException {
    // constructor methods

    /**
     * Default constructor method. The code used should be one of the constants
     * defined in the DOMException interface. The String message can be used to
     * explain the exception in more detail.
     * @see org.w3c.dom.DOMException */
    public DDOMException(short code, String message) {
        super(code, message);
    }
}

Class ddom.dom.DDOMImplementation

package ddom.dom;

import org.w3c.dom.DOMImplementation;

/**
 * The class DDOMImplementation represents the DOM implementation used by
 * DDocument and DDocumentBuilder. It allows to check which optional features
 * are supported by this implementation. For details on DOM implementations
 * please check the documentation of the DOMImplementation interface.
 * @see org.w3c.dom.DOMImplementation *@
 * @Author Mathias Neum&ouml;ller
 * @Version 1.0 */

public class DDOMImplementation implements DOMImplementation {
    // constructor methods

    /** Default constructor. */
    public DDOMImplementation() {}

    // implementation of DOMImplementation methods

    /**
     * Tests if the DOM implementation implements a specific feature.
     * Note: The DOM implementation does currently not support any extended
     * features, therefore this method will always return false.
     */
    public boolean hasFeature(String feature, String version) {
        return false;
    }
}

Class ddom.dom.Dictionary

package ddom.dom;
```
import java.util.List;
import java.util.ArrayList;

/**
 * The class Dictionary contains the dictionaries used to store the information
 * contained in a DOM document. The data structure extends itself to the
 * required size. Data appended to a dictionary is compared with the current
 * content of it. If the data already exists in that dictionary, a reference to
 * the current entry is returned. Otherwise a new entry is created in the
 * dictionary and a reference to the new entry is returned.
 */
@see DDocument
@author Mathias Neumüller
@version 1.0
*/
public class Dictionary {
    private static final int DOMAIN_INCREMENT = 10;
    // declaration of variables
    private List[][] data;
    // constructors methods
    /**
     * Default constructor. */
    Dictionary() {
        // currently 12 different types of entries
        // only one domain per type as most types do not support multiple domains
        data = new List[13][1];
    }
    // implementation of Dictionary methods
    /**
     * Returns the entry value at given index in the dictionary of the given type.
     */
    public String get(byte type, int index) {
        return get(type, DNode.NO_DOMAIN, index);
    }
    /**
     * Returns the entry value at given index in the dictionary of the given type
     * and domain.
     */
    public String get(byte type, int domain, int index) {
        if (index == DNode.NO_DATA) {
            return "";
        } else {
            String s;
            try {
                s = (String) data[alias(type)][domain].get(index - 1);
            } catch (Exception e) {
                s = e.toString();
            }
            return s;
        }
    }
}
C. Source code

* Searches for an entry with given type and value in the dictionary.
* If the entry exists, its index within the dictionary is returned. If no
* such entry exists, the constant Node.NO DATA is returned.

```java
public int find(byte type, String value) {
    return find(type, DNode.NO DOMAIN, value);
}
```

*/

* Searches for an entry with given type, domain and value in the dictionary.
* If the entry exists, its index within the dictionary is returned. If no
* such entry exists, the constant Node.NO DATA is returned.

```java
public int find(byte type, int domain, String value) {
    if (value != null) {
        if (domain >= data[type].length) return DNode.NO DATA;
        if (data[type][domain] == null) return DNode.NO DATA;
        int index = data[type][domain].indexOf(value);
        if (index == -1) {
            return DNode.NO DATA;
        } else {
            return index + 1;
        }
    }
    else {
        return DNode.NO DATA;
    }
}
```

*/

* This method enters an entry to the dictionary and returns a reference to
* it. It is used for domainless types like element names and will enter its
* entries in the special domain `<CODE:NO DOMAIN</CODE>` for the specified
* type.

* @see Dictionary#append(byte, String)

```java
public int append(byte type, String value) {
    return append(type, DNode.NO DOMAIN, value);
}
```

*/

* This method enters an entry to the dictionary and returns a reference to
* it. If an entry with the specific value already exists in the specified
* domain of the specified type, a reference to this entry is returns. If
* the entry doesn’t exist, a new entry is created and a refence to this
* entry is returned.

*/

```java
public int append(byte type, int domain, String value) {
    if (value != null) {
        if (domain >= data[type].length) growDomainSpace(type, domain);
        if (data[type][domain] == null) data[type][domain] = new ArrayList();
        int index = find(type, domain, value);
        if (index == DNode.NO DATA) {
            index = data[type][domain].size();
            data[type][domain].add(index, value);
            return index + 1;
        } else {
            return index;
        }
    } else {
        return DNode.NO DATA;
    }
```
This method dumps the content of the contained dictionaries to a
<CODE>PrintWriter</CODE>. This method is included for debug purposes
only and should not be used by normal applications. It is made public
* to allow debugging from outside the package.

```java
public void dump(java.io.PrintWriter writer) {
    for (int type = 0; type < data.length; type++) {
        for (int domain = 0; domain < data[type].length; domain++) {
            if (data[type][domain] != null) {
                writer.println("Dictionary for type \"" +
                                + type + ", domain \"" +
                                + domain + ": ");
                writer.println("="
                                + data[type][domain].toString().trim());
                writer.println();
            }
            writer.flush();
        }
    }
}
```

// private methods

```java
private void growDomainSpace(byte type, int domain) {
    int newsize = data[type].length;
    while (domain >= newsize) newsize = newsize + DOMAIN_INCREMENT;
    List[] domainSpace = new List[newsize];
    for (int i = 0; i < data[type].length; i++) domainSpace[i] = data[type][i];
    data[type] = domainSpace;
}
```

```java
private byte alias(byte type) {
    if (type < 0) return (byte) -type; else return type;
}
```

**Class ddom.dom.DParser**

```java
package ddom.dom;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.w3c.dom.*;
```
C. Source code

The class DParser is used to parse a org.xml.sax.InputSource into a compressed DOM document. Internally a SAX Parser is used to generate the required structure, but this may change in the future for performance reasons. Currently only Elements, Text nodes and Processing Instructions are handled properly. Attributes are handled correctly as long as they only represent a single textual value that can be resolved using the underlying SAX parser. Other elements are either not recognised by the SAX parser, like comments, or not handled by the current DOM implementation, like Notations.

@see ddom.jaxp/DDocumentBuilderFactory
@see ddom.jaxp/DDocumentBuilder

@see Mathias Neumann\11er
@version 1.0

public class DParser extends HandlerBase
{

    // declaration of instance variables

    private SAXParser parser;
    private Document doc;
    private Node parent;
    private String lastText;
    private java.util.Stack stack;
    private boolean ignoreWhiteSpace;

    // constructor methods

    /**
     * Creates a new parser object, ignoring whitespace in elements-only nodes.
     */
    public DParser() throws SAXException, ParserConfigurationException {
        this(true);
    }

    /**
     * Creates a new parser object. Whitespace contained in elements that can
     * only have elements as content is ignored if the boolean parameter is
     * set to true.
     */
    public DParser(boolean ignoreWhiteSpace)
            throws SAXException, ParserConfigurationException {
        System.setProperty("javax.xml.parsers.SAXParserFactory",
                "org.apache.xerces.jaxp.SAXParserFactoryImpl");
        parser = SAXParserFactory.newInstance().newSAXParser();
        stack = new java.util.Stack();
        doc = null;
        parent = null;
        lastText = null;
        this.ignoreWhiteSpace = ignoreWhiteSpace;
    }

    // implementation of DParser methods

    /**
     * Parses a input source and returns the parsed document.
     * Note that the returned document is read-only, i.e. no nodes can be added
     * or changed after parsing. This limitation may be lifted in the future with
     * less restrictive versions of the DOM.
     */
    public Document parse(InputSource input)
            throws org.xml.sax.SAXException,
java.io.IOException, ParserConfigurationException
{
    doc = null;
    parent = null;
    lastText = null;
    stack.clear();
    parser.parse(input, this);
    return doc;
}

/**
 * Releases the reference to the buffered document. This is done in order to
 * allow the garbage collector to reclaim the memory once no more external
 * references to the last parsed document exist.
 */
public void clearDocumentBuffer() {
    doc = null;
}

/**
** Handles Processing instruction events. */
public void processingInstruction(String target, String data)
{
    appendLastText();
    parent.appendChild(doc.createElementProcessingInstruction(target, data));
} // processingInstruction(String,String)

/**
** Handles the Start document event. */
public void startDocument()
{
    doc = new DDocument();
    parent = doc;
} // startDocument()

/**
** Handles Start element events. */
public void startElement(String name, AttributeList attrs)
{
    appendLastText();
    stack.push(parent);
    parent = doc.createElement(name);
    if (attrs != null)
    {
        int len = attrs.getLength();
        Element e = (Element) parent;
        for (int i = 0; i < len; i++)
        {
            e.setAttribute(attrs.getName(i), attrs.getValue(i));
            /**<@todo: Add support for multiple value attributes here?*/
        }
    }
} // startElement(String,AttributeList)

/**
** Handles Characters events. */
public void characters(char ch[], int start, int length)
{
    if (lastText == null) {
        lastText = new String(ch, start, length);
    } else {
        lastText = lastText.concat(new String(ch, start, length));
    }
} // characters(char[],int,int);

/**
** Handles Ignorable whitespace events. */
public void ignorableWhitespace(char ch[], int start, int length)
{
    if (!ignoreWhiteSpace) characters(ch, start, length);
} // ignorableWhitespace(char [], int, int);

/** Handles End element events. */
public void endElement(String name)
{
    appendLastText();
    Node lastparent = (Node) stack.pop();
    lastparent.appendChild(parent);
    parent = lastparent;
} // endElement(String)

/** Handles the End document event. */
public void endDocument()
{
    parent = null;
    stack = null;
    ((DDocument) doc).close();
} // endDocument()

/*
 * Used to append the content of the last text node before anything else
 * is appended. The reason for this is to allow multiple following text
 * nodes to be joined together prior to appending.
 */
void appendLastText()
{
    if (lastText != null) {
        parent.appendChild(doc.createTextNode(lastText));
        lastText = null;
    }
}

// / ErrorHandler methods

/** Handles Warnings. */
public void warning(SAXParseException ex)
{
    System.err.println("[Warning]" + getLocationString(ex) + ":" + ex.getMessage());
}

/** Handles Errors. */
public void error(SAXParseException ex)
{
    System.err.println("[Error]" + getLocationString(ex) + ":" + ex.getMessage());
}

/** Handles Fatal errors. */
public void fatalError(SAXParseException ex)
throws SAXException
{
    System.err.println("[Fatal Error]" + getLocationString(ex) + ":" + ex.getMessage());
    throw ex;
}
C. Source code

```java
private String getLocationString(SAXParseException ex) {
    StringBuffer str = new StringBuffer();
    String systemId = ex.getSystemId();
    if (systemId != null) {
        int index = systemId.lastIndexOf('/');
        if (index != -1) {
            systemId = systemId.substring(index + 1);
            str.append(systemId);
        }
        str.append(':');
        str.append(ex.getLineNumber());
        str.append(':');
        str.append(ex.getColumnNumber());
    }
    return str.toString();
}
```

/** Returns a string of the location. */

```java
public boolean isValidating() {
    return parser.isValidating();
}
```

JAXP integration (package ddom.jaxp)

Class ddom.jaxp-DDocumentBuilderFactory

```java
package ddom.jaxp;

import javax.xml.parsers.*;

/** The class DDocumentBuilderFactory is a factory class for DOM
 * DocumentBuilders. Its main method, newDocumentBuilder(), is used to generate
 * DDocumentBuilders. */

/** Note: To use the DOM implementation in a JAXP compliant application,
 * simply set the system property "javax.xml.parsers.DocumentBuilderFactory" to
 * the name of this class, i.e. "ddom.jaxp.DDocumentBuilderFactory". You can
 * then create Documents as with any other DOM implementation, e.g./P>

 * "ddom.jaxp.DDocumentBuilderFactory");
 * DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
 * DocumentBuilder builder = factory.newDocumentBuilder();
 * Document document = builder.newDocument();

/** @see javax.xml.parsers.DocumentBuilderFactory
 * @see ddom.jaxp.DDocumentBuilder */

@author Mathias Neumüller
@version 1.0
```
package ddom.jaxp;

import java.util.Stack;
import ddom.dom.*;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.xml.sax.*;

public class DDocumentBuilderFactory extends DocumentBuilderFactory {

    // constructor methods

    /** Default constructor. */
    public DDocumentBuilderFactory() {

        // implementation of DocumentBuilderFactory methods

        /**
         * Returns the value of the parser factory attribute with the given name.
         * As the DDOM implementation currently doesn’t support any extended functionality, this method will always throw an exception.
         * @exception java.lang.IllegalArgumentException in case of unknown attributes
         *
         */
        public Object getAttribute(String att) throws java.lang.IllegalArgumentException {
            throw new java.lang.IllegalArgumentException("DDocumentBuilder_error:_Attribute_:"+att+"_not_recognised.");
        }

        /**
         * Returns a new DDOM DocumentBuilder object. Note that you have
         * to use this method, as the class DDocumentBuilder does not provide a public
         * constructor.
         */
        public DocumentBuilder newDocumentBuilder() throws javax.xml.parsers.ParserConfigurationException {
            try {
                return new DDocumentBuilder(this.isIgnoringElementContentWhitespace());
            } catch (org.xml.sax.SAXException e) {
                throw new ParserConfigurationException(e.getMessage());
            }
        }

        /**
         * Sets the value of the parser factory attribute with the given name to the
         * given value. As the DDOM implementation currently doesn’t support any
         * extended functionality, this method will always throw an exception.
         * @exception java.lang.IllegalArgumentException in case of unknown attributes
         *
         */
        public void setAttribute(String att, Object parm2) throws java.lang.IllegalArgumentException {
            throw new java.lang.IllegalArgumentException("DDocumentBuilder_error:_Attribute_:"+att+"_not_recognised.");
        }
    }
}

Class ddom.jaxp.DDocumentBuilder
C. Source code

/**
 * The class DDocumentBuilder is a factory class for DDOM Documents.
 * Its two main methods <CODE>newDocument()</CODE> and <CODE>parse()</CODE> can be used to generate empty documents or parse XML from a given <CODE>InputSource</CODE> respectively. Note that this class cannot be instantiated directly as its constructor method is declared protected.
 * In order to get an instance of this class use the appropriate method of its factory class, <CODE>DDocumentBuilderFactory.getDocumentBuilder()</CODE>.
 * @see DDocumentBuilderFactory#newDocumentBuilder
 * @see org.w3c.dom.Document
 * @author Mathias Neumüller
 * @version 1.0
 */

public class DDocumentBuilder extends DocumentBuilder {

    // declaration of instance variables
    /** Reference to the parser used. */
    DParser parser;

    // constructor methods
    /** Default constructor. */
    protected DDocumentBuilder(boolean ignoreWhiteSpace)
    throws SAXException, ParserConfigurationException {
        parser = new DParser(ignoreWhiteSpace);
    }

    // implementation of DocumentBuilder methods
    /**
     * Returns a DOMImplementation object representing the DDOM implementation.
     */
    public DOMImplementation getDOMImplementation() {
        return new DDOMImplementation();
    }

    /**
     * Returns if this DocumentBuilder is aware of XML namespaces.
     * DDOM is currently a DOM level 1 implementation and is therefore not namespace aware. This function call will always return false.
     */
    public boolean isNamespaceAware() {
        return false;
    }

    /**
     * Parses the given InputSource to a DDOM document. If an exception occurs during the parsing process, this exception is thrown up to the calling program in form of a SAXException. This is due to the fact, that the parser internally uses a SAX parser to parse the document. The document returned is a read-only document. Any try to modify its content will result in an exception being raised.
     * @exception org.xml.sax.SAXException any exception occurring during the parsing process
     */
    public Document parse(InputSource input)
    throws org.xml.sax.SAXException, java.io.IOException {

/*
C. Source code

```java
try {
    Document doc = parser.parse(input);
    parser.clearDocumentBuffer();
    return doc;
} catch (ParserConfigurationException e) {
    throw new SAXException(e);
}
```

/** *
* Sets the entity resolver for the parsing process to the given class.
* By default the internal entity resolver of the parser is used.*/

/** * Note:<EM> This functionality is currently not supported by the DOM * implementation. Calls upon this method will always result in an exception * being raised.</EM> */

public void setEntityResolver(EntityResolver parm1) {
    /**@todo: implement this javax.xml.parsers.DocumentBuilder abstract method*/
    throw new java.lang.UnsupportedOperationException("Method setEntityResolver() not yet implemented.");
}

/** *
* Returns if this DocumentBuilder is validating the XML code. The result *
* of this call will depend on the capabilities of the underlying SAX parser. *
*/

public boolean isValidating() {
    return parser.isValidating();
}

/** *
* Returns a new, empty DOM document.
* The document returned has read–write access.
*/

public Document newDocument() {
    return new DDocument();
}

/** *
* Sets the error handler for the parsing process to the given class.
* By default the internal error handler of the parser is used.*/

/** * Note:<EM> This functionality is currently not supported by the DOM * implementation. Calls upon this method will always result in an exception * being raised.</EM> */

/** @exception java.lang.UnsupportedOperationException always */

public void setErrorHandler(ErrorHandler parm1) {
    /**@todo: implement this javax.xml.parsers.DocumentBuilder abstract method*/
    throw new java.lang.UnsupportedOperationException("Method setErrorHandler() not yet implemented.");
}
```
Test programs (package ddom)

Class ddom.DTreeTester

package ddom;

import java.io.File;
import javax.xml.parsers.∗;
import org.w3c.dom.Document;

/∗<P>The class DTreeTester uses the JAXP API to parse an XML document into a
DOM tree. The memory consumed for the document is measured and displayed.</P>
<P>Note: Ignorable whitespace is skipped and not added to the tree if the
DOM implementation supports this feature and a validating parser is used.</P>∗

@version 1.0

public class DTreeTester {

// declaration of instance variables
protected Document doc;
protected DocumentBuilder builder;

// constructor methods

∗∗
∗ Default constructor, creates a DTreeTester for the DOM implementation.
∗
∗ /
public DTreeTester() {
    this("ddom.jaxp/DDocumentBuilderFactory", false);
}

∗∗
∗ The constructor creates a new DocumentBuilder that is later used for
∗ parsing.
∗
∗ /
public DTreeTester(String DocumentBuilderFactoryName, boolean ignore) {
    System.setProperty("javax.xml.parsers.DocumentBuilderFactory", DocumentBuilderFactoryName);
    doc = null;
    try {
        DocumentBuilderFactory fac = DocumentBuilderFactory.newInstance();
        fac.setIgnoringElementContentWhitespace(ignore);
        builder = fac.newDocumentBuilder();
    }
    catch (Exception e) {
        e.printStackTrace();
        System.exit(-1);
    }
}

// implementation of DTreeTester methods

∗∗
∗ Function to calculate the amount of memory used.
∗ The memory used is calculated as difference between total and free memory
∗ of the virtual machine. The garbage collector is run immediately before the
∗ measurement is performed to get rid of any unused object still in memory.
"}
C. Source code

```java
public static long getUsedMemory() {
    Runtime rt = Runtime.getRuntime();
    rt.gc();
    return rt.totalMemory() - rt.freeMemory();
}

/** Parses the specified xml-file using the DocumentBuilder */
public void parse(String filename) {
    try {
        doc = builder.parse(new File(filename));
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(-1);
    }
}

/** The main method used to test the memory consumption of a DOM implementation. A DocumentBuilder is created using the JAXP interface. Then the memory available is measured. After the document was parsed, the measurement is repeated and the difference printed out as the amount of memory used for the document. */
public static void main(String[] args) {
    if (args.length < 1) {
        System.err.println("DTreeTester: DOM tree test utility");
        System.err.println("Usage: java ddom.DTreeTester xml-file");
    } else {
        DTreeTester test = new DTreeTester("org.apache.xerces.jaxp.DocumentBuilderFactoryImpl", false);
        "ddom.jaxp.DocumentBuilderFactory", false);
        long before = getUsedMemory();
        test.parse(args[0]);
        long after = getUsedMemory();
    }
}

Class ddom.XML2JTree

package ddom;

// Import the W3C DOM classes
import org.w3c.dom.*;

// We are going to use JAXP's classes for DOM I/O
import javax.xml.parsers.*;

// Import other Java classes
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
```

import java.io.*;
import java.net.*;

/**
 * The class XML2JTree displays XML in a JTree.
 * This class is taken from chapter 3 "the DOM" of the book
 * "Professional Java XML". It was modified to use the DDOM implementation
 * rather than the default JAXP DOM parser and added to the package ddom for
 * convenience. No further modifications were necessary.
 * @author John Davies (modified by Mathias Neumüller)
 * @version 1.0
 */
public class XML2JTree extends JPanel {
    private JTree jTree;
    private static JFrame frame;

    public static final int FRAME_WIDTH = 440;
    public static final int FRAME_HEIGHT = 280;

    public XML2JTree(Node root, boolean showDetails) {
        // Take a DOM and convert it to a Tree model for the JTree
        DefaultMutableTreeNode top = createTreeNode(root, showDetails);
        DefaultTreeModel dtModel = new DefaultTreeModel(top);

        // Create our JTree
        jTree = new JTree(dtModel);

        // We have no tree listener but this looks more natural
        jTree.setSelectionModel().setSelectionMode(      
            TreeSelectionModel.SINGLE_TREE_SELECTION);
        jTree.setShowsRootHandles(true);

        // If this were editable it would result in too much code
        // for this demo.
        jTree.setEditable(false);

        // Create a new JScrollPane but override one of the methods.
        JScrollPane jScroll = new JScrollPane()
        {
            // This keeps the scrollpane a reasonable size
            public Dimension getPreferredSize()
            {
                return new Dimension(FRAME_WIDTH - 20, FRAME_HEIGHT - 40);
            }
        }

        // Wrap the JTree in a JScrollPane so that we can scroll it in the JScrollPane.
        jScroll.setViewport().add(jTree);

        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.add("Center", jScroll);
        add("Center", panel);
    }

    /**
     * This takes a DOM Node and recursively through the children until each one is
     */

added to a DefaultMutableTreeNode. This can then be used by the JTree as a
tree model. The second parameter can be used to provide more visual detail
for debugging.

```java
/*
 * protected DefaultMutableTreeNode
 * createTreeNode( Node root, boolean showDetails )
 * {
 *     DefaultMutableTreeNode dmtNode = null;
 *     String type = getNodeType(root);
 *     String name = root.getNodeName();
 *     String value = root.getNodeValue();
 *     if( showDetails )
 *     {
 *         dmtNode = new DefaultMutableTreeNode("["+type+"]−−>"+name+"\=\"+value);
 *     }
 *     else
 *     {
 *         // Special case for TEXT NODE, others are similar
 *         // but not catered for here.
 *         dmtNode = new DefaultMutableTreeNode(
 *             root.getNodeType() == Node.TEXT_NODE ? value : name);
 *     }
 *     // Display the attributes if there are any
 *     NamedNodeMap attrs = root.getAttributes();
 *     if( attrs != null && showDetails )
 *     {
 *         for( int i = 0; i < attrs.getLength(); i++ )
 *         {
 *             Node attNode = attrs.item(i);
 *             String attName = attNode.getNodeName().trim();
 *             String attValue = attNode.getNodeValue().trim();
 *             if(attValue != null)
 *             {
 *                 if (attValue.length() > 0)
 *                 {
 *                     dmtNode.add(new DefaultMutableTreeNode("[Attribute]−−>\"+attNames+"\=\"+attValue+\"\"));
 *                 }
 *             }
 *         }
 *     }
 *     // If there are any children and they are non-null then recurse...
 *     if(root.hasChildNodes())
 *     {
 *         NodeList childNodes = root.getChildNodes();
 *         if(childNodes != null)
 *         {
 *             for (int k=0; k<childNodes.getLength(); k++)
 *             {
 *                 Node nd = childNodes.item(k);
 *                 if( nd != null )
 *                 {
 *                     // A special case could be made for each Node type.
 *                     if( nd.getNodeType() == Node.ELEMENT_NODE )
 *                     {
 *                         // Do something with the element node.
 *                     }
 *                 }
 *             }
 *         }
 *     }
 */
```
dmtNode.add(createTreeNode(nd, showDetails));

    } // This is the default
  String data = nd.getNodeValue();
  if(data != null)
  {
    data = data.trim();
    if(!data.equals("\n") && !data.equals("\r\n") &&
       data.length() > 0)
    {
      dmtNode.add(createTreeNode(nd, showDetails));
    }
  }
  
  } }
  return dmtNode;
}
/**
This simple method returns a displayable string given a NodeType
*/

public String getNodeType( Node node )
{
  String type;

  switch( node.getNodeType() )
  {
    case Node.ELEMENT_NODE:
    {
      type = "Element";
      break;
    }
    case Node.ATTRIBUTE_NODE:
    {
      type = "Attribute";
      break;
    }
    case Node.TEXT_NODE:
    {
      type = "Text";
      break;
    }
    case Node.CDATA_SECTION_NODE:
    {
      type = "CDATA section";
      break;
    }
    case Node.ENTITYREFERENCE_NODE:
    {
      type = "Entity reference";
      break;
    }
    case Node.ENTITY_NODE:
    {
      type = "Entity";
      break;
    }
  
  }
C. Source code

```java
private static void exit()
{
    System.out.println("Graceful exit");
    System.exit(0);
}

private static void main( String[] args )
{
    Document doc = null;

    // Just check we have the right parameters first.
```
if ( args.length < 1 )
{
    System.out.println("Usage: java ddom.XML2JTree filename.xml");
    return;
}
String filename = args[0];

boolean showDetails = false;
if ( args.length == 2 )
{
    // If anything's there override the default
    showDetails = Boolean.valueOf(args[1]).booleanValue();
}

// Create a frame to "hold" our class
frame = new JFrame("XML to JTree");
Toolkit toolkit = Toolkit.getDefaultToolkit();
Dimension dim = toolkit.getScreenSize();
int screenHeight = dim.height;
int screenWidth = dim.width;

// This should display a WIDTH x HEIGHT sized Frame in the middle
// of the screen
frame.setBounds((screenWidth-FRAME_WIDTH)/2,
   (screenHeight-FRAME_HEIGHT)/2, FRAME_WIDTH, FRAME_HEIGHT);
frame.setBackground(Color.lightGray);
frame.getContentPane().setLayout(new BorderLayout());

// Give our frame an icon when it's minimized.
frame.setIconImage(toolkit.getImage("Wrox.gif"));

// Add a WindowListener so that we can close the window
WindowListener wndCloser = new WindowAdapter()
{
    public void windowClosing(WindowEvent e)
    {
        exit();
    }
};
frame.addWindowListener(wndCloser);

try
{
        "ddom.jaxp/DDocumentBuilderFactory");
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    dbf.setValidating(false); // Not important for this demo
    DocumentBuilder db = dbf.newDocumentBuilder();
    doc = db.parse(filename);
}
catch( FileNotFoundException fnfEx )
{
    // Display a "nice" warning message if the file isn't there.
    JOptionPane.showMessageDialog(frame, filename+" was not found",
        "Warning", JOptionPane.WARNING_MESSAGE);
    System.out.println();
    exit();
}
```java
} catch (Exception ex) {
    JOptionPane.showMessageDialog(frame, ex.getMessage(), "Exception",
        JOptionPane.WARNING_MESSAGE);
    ex.printStackTrace();
    exit();
}

Node root = (Node) doc.getDocumentElement();

frame.getContentPane().add(new XML2JTree(root, showDetails), BorderLayout.CENTER);
frame.validate();
frame.setVisible(true);
```