Implementing a subset of DFDL

A basic parser for a subset of the DFDL specification and related libraries

Yi ZHU

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2005

Authorship declaration

I, Yi Zhu, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understand the penalties associated with plagiarism.

Signed:

Date: August 26, 2005

Matriculation no: 0457001

Acknowledgements

Many people helped in the development of my project during the last three months, and I am grateful to them all. Neil Chue Hong and Tom Sugden, who are my supervisors, gave me huge help within the range of possibilities. Neil directed the high-level orientation and guidance for the project and gave me a lot of valuable suggestions on the dissertation. Tom, helped me with my detailed design, coding and English presentation, and he was very kind and patient. Moreover, they introduced me to Martin Westhead, who was the author of “GGF DFDL Primer”. Martin explained some of the vague concepts of DFDL for me and also gave me some significant suggestions. Besides, the DFDL working group was also quite helpful, especially Susan Malaika and Kristoffer H. Rose who helped me to understand the DFDL at the first stage and provided me with some original ideas of the project. For instance, Susan suggested that the JPEG data format would be an interesting topic to be described, and Kristoffer provided me with several detailed examples of how to describe data files using DFDL.

Abstract

DFDL is a kind of XML-based data format description language, which can be used to describe the structure of any data file for data interchange in the Grid environment. Since DFDL is still under discussion, I chose to implement a subset of the DFDL specification and determined how to describe the JPEG data format based on it. After that, I designed a library, including a generic DFDL description parser and a data file parser, which can be used to convert any data file to a XML representation based on the corresponding DFDL description. The whole project follows formal software engineering principles, from system analysis and design to the implementation. During the coding, I utilised unit testing to ensure that all core modules worked correctly; and then I tested the entire system using a simple binary data file, a simple text file and a JPEG image with their corresponding DFDL descriptions.

This dissertation is organised in a similar structure. I explain the basic concept of the DFDL and the JPEG data format at first; and then discuss how I analysed the entire project, including requirement analysis and risk analysis etc., followed by the system design and implementation as well as the testing.

Table of Contents

Chapter 1 Introduction

1.1 Background

From the early 1970s when computers were first linked by networks, people have been thinking about the idea of simultaneously making use of different computational resources and harnessing unused CPU cycles. In the 1990s, distributed computing scaled to a global level as the Internet and high-performance physical interconnection technology matured, and then a brand new concept, the Grid, occurred. [1]

The Grid [2] suggests a computing framework similar to the electric power grid, which provides power for consumers all over the nation, even around the world, without the requirement to know where the electricity comes from and how it is generated. Similarly, the Grid intends to provide users with distributed computational power, and they do not need to know where their jobs are running and what kind of machine they are using.

From about 2000, Web Services emerged from the business world in an attempt to establish a framework for machine-to-machine communication using XML-based technologies as the basis for language neutral and machine independent communications. At roughly the same time, the concept of Web Services was introduced into the new generation of the Grid, the Open Grid Services Architecture (OGSA), which defined the Grid architecture in terms of services: heterogeneous resources on the Grid are accessed via XML Web Services using the Web Services Resource Framework (WS-RF).

Although different Grid applications have different characteristics, there are some challenges that are common to the Grid. Firstly, Security Challenges, which concern problems like “Is the user who they say they are?” and “Is the user allowed to do what they are requesting?” Secondly, Scheduling Challenges, which consider issues such as how to find the appropriate resources and how tasks are directed to that resources. Last but not least, Data Challenges: heterogeneous data resources stored in different formats are distributed across the Grid, and it is hard to make efficient and transparent use of these data sets.

However, there are key efforts underway to tackle these challenges and define standards that would allow the easy pooling and sharing of all computing resources, including cycles, storage, and data in a way that can promote mass adoption of grid computing [1]. For example, the OGSA-DAI project [3] is the leading effort for the Data Challenges. It develops middleware, which is a part of the Data Management component in the Globus Toolkit 4, to allow data resources, currently mainly database systems, to be accessed and integrated into a Grid environment.

For using OGSA-DAI, a mechanism is needed to specify different data formats in a universal manner, and DFDL is such a means. As DFDL is still under research and discussion, it is very helpful to try to implement a subset of it, including both parsers and high-level applications, which is also the aim of this project.

1.2 DFDL

DFDL is short for Data Format Description Language, which is an XML-based language used for describing the structure of binary and character encoded files and data streams so that their format and structure can be exposed [4]. It can be used to represent heterogeneous data resources in the Grid environment, currently mainly database systems and flat binary files. This effort specifically does not aim to create a generic data representation language. Rather, DFDL endeavors to describe existing formats in an actionable manner that makes the data in its current format accessible through generic mechanisms.

The DFDL description would sit in a logically separate file from the data itself. The description would provide a hierarchical description that would structure and semantically label the underlying bits stream. It would specify how bits are to be interpreted as parts of low-level data types, such as integers, floats, strings; how low-level types are assembled into scientifically relevant forms such as arrays; how meaning is assigned to these forms through association with variable names and metadata such as units; and how arrays and the overall structure of the binary file are parameterized based on array dimensions, flags specifying optional file components, etc. Any DFDL described file can then be accessed, queried and integrated, regardless of its data format.

Figure 1 below illustrates a situation when DFDL may be useful. There are some legacy data about protein folding stored in a flat binary data file in China, meanwhile in UK there is another data set about protein folding, which is a text file. A biologist in US is going to synthesize new information form both these two data resources. How can the user simultaneously deal with these two heterogeneous data formats? Here DFDL is a good choice. Along with each data resource, there is a DFDL description describing the data structure, and the user can then use a parser to understand the data format that DFDL described and some libraries to do integrated queries.

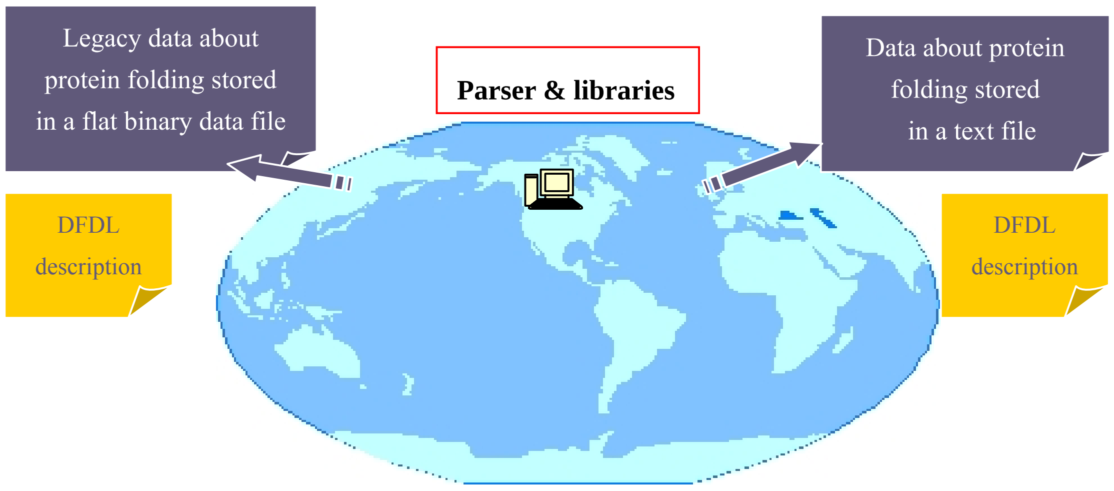

Legacy data about protein folding stored in a flat binary data file
Parser & libraries
Data about protein folding stored in a text file
DFDL description
DFDL description

Figure 1 A simple scenario for DFDL

When using DFDL, the format of data in a data stream is described by means of a DFDL Schema, which is an XML Schema containing only a subset of the constructs available in full XML Schema Description Language, and augmented with special DFDL annotations that carry the information about the data format or representation. (see Chapter 2 for more details)

1.3 XML & XML schema

As mentioned before, DFDL is based on XML, and can be thought of as a combination of XML schema and DFDL annotations.

XML (Extensible Markup Language) is a standard for creating markup languages which describe the structure of data. It is a metalanguage, or in other words, a language for describing and defining languages. It provides an essential mechanism for transferring data between services in an application and platform neutral format. However, the drawback of using it is the large overhead that XML tagging imposes, especially when the dataset is huge. Furthermore, there are large amounts of valuable legacy datasets which are not stored in XML format, having existed prior to XML. The DFDL working group at the Global Grid Forum (GGF) therefore intended to take advantages of various characteristics of XML and define a generic description language for data.

XML has some features that are valuable for the DFDL. For instance, XML allows authors to define their own tags, providing a mechanism for describing data in an environment where a human and an application both can determine what the data is and what it is representing. As DFDL is based on XML, it is both human-legible and computer-legible. On the other hand, XML is an extensible and self-descriptive language, and theoretically, it is a Turing-Complete language which could generate any type of languages without limitation. Therefore, in theory, DFDL can be used to define any type of data format with the help of XML schema, although there are still some problems that have not been solved.

The purpose of an XML Schema is to define the legal building blocks of an XML document. It defines elements and attributes that can appear in a document, the data types for them and the order of them, as well as their default or fixed values. The DFDL user describes the abstract data model using an XSD, and the XSD-based model is then augmented by the so-called DFDL annotations specifying how the data model is represented in an underlying text or binary form. Hence, data that is described by DFDL can be thought of as if it were XML data even though the representation is a much smaller and more efficient one. This concept is generally called "tagless XML". It is anticipated that an XSD approach will make it possible to construct DFDL APIs similar to DOM and SAX and to simplify exporting of DFDL described data to XML documents [5].

1.4 JPEG

In practice, DFDL can be used to describe databases, including both relational databases (Oracle, MS SQL Server, MySQL, etc.) and XML databases (Xindice, eXist, etc.), and all kinds of flat binary data files, even text documents and pictures. However, in the context of my project, I intend to use a data format which has a clear structure and is meaningful to be described. After considering pros and cons of several possible data formats, I decided to use JPEG as the candidate data format to be described using DFDL for analysis and testing during my project.

JPEG is an international standard for colour image compression, defining a group of compression algorithms and many coding alternatives for continuous-tone, still images. However, it does not define the meaning or format of the components that comprise the image. Attributes like the color space and pixel aspect ratio must be specified out-of-band with respect to the JPEG bits stream. The JPEG File Interchange Format (JFIF) is a defacto standard that provides this extra information using an application marker segment (APP0, APP1 etc.) [6]. It enables JPEG bit streams to be exchanged between a variety of platforms and applications [7]. Therefore, usually when people refer to "JPEG", they actually mean JPEG JFIF. In my project, I will focus on the JPEG JFIF data format defined in ISO DIS 10918-1 [8] and JPEG File Interchange Format Specification. Furthermore, in this article, I also use "JPEG" to stand for JPEG JFIF.

I chose JPEG as the underlying data format because, firstly, it is a very popular data format and used in almost every field ranging from peoples' daily lives to scientific research: there exist vivid pictures of harmonious families, as well as huge images of galaxies. These pictures are distributed all over the world and are quite likely to be shared and transferred over the Grid. Secondly, JPEG files have clear structures that can be identified by different markers. However, here clear structures do not mean simple or less meaningful. In opposite, different segments nest with each other and form a very complex multiple layers format, and meanwhile these structures contains some particular attributes, all of which are valuable topics to explore whether DFDL is powerful enough to handle them. For instance, the occurrence and the length of a segment may be determined by the contents of one or more other segments, requiring some calculation and even conditional distinction; and sometimes JPEG uses “nibble” type data items which would be a big challenge for DFDL to express (see Chapter 3 for more details). Thirdly, JPEG files include not only the data layout information, but also some meaningful contents, such as colours and compression algorithms, which are interesting topics to be described.

What’s more, this format was also suggested by the GGF DFDL Working Group because there are still few real-world examples of DFDL implementation and describing JPEG would be a brand new experiment; at the same time, examining whether DFDL can cope with those “dependence” problems of JEPG may result in some other useful comments. So the result of my project will be valuable for the entire DFDL project.

1.5 Project overview

In a nutshell, the aim of this project is to determine how to describe the JPEG data format using a subset of DFDL specification, and subsequently design and implement a generic parser for this subset. The parser should be capable of understanding not only the DFDL description of JPEG but also other DFDL descriptions based on the same subset that has been focused on. Finally, some related libraries, which produce an XML representation of the original data file utilising this parser. (see Figure 2)

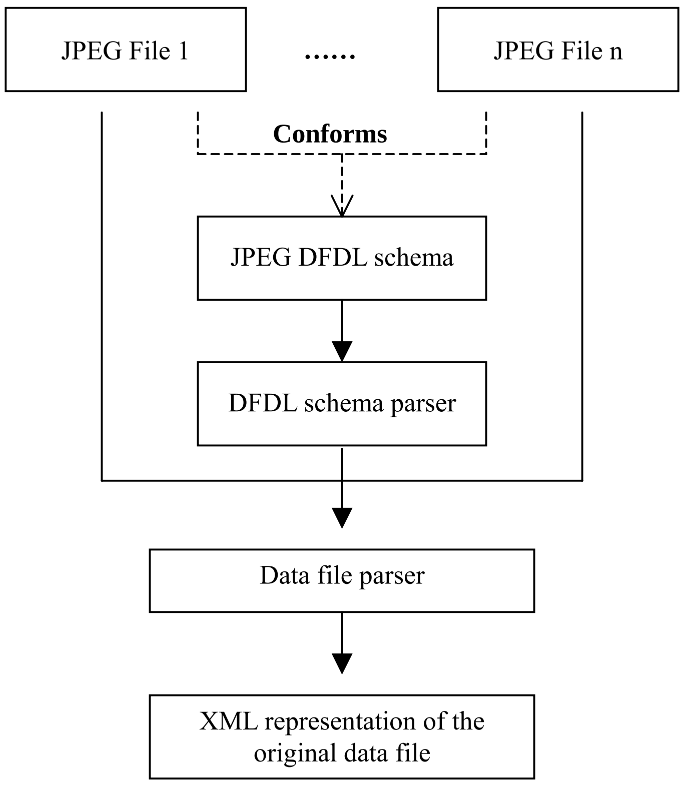


Figure 2 High-level design for the entire project

In addition, for increasing the chances of project success and ensuring good usability and efficient data transfer through the system, I need to follow effective software engineering principles and produce a good design.

The goal of the project is to utilise a subset of DFDL to solve a real-world problem – describing the JPEG data format – and to see whether DFDL is capable of doing that, including discussing any shortage or limitations; and then to explore potential difficulties in parsing DFDL schemas, by actually implementing a generic parser.

Chapter 2 DFDL

2.1 Architecture

DFDL is a descriptive language, describing the data representation in a separate file. When an application wants to access the data, it is via parsers that read a data file or stream of raw data along with the associated DFDL description. The parser makes various data formats understandable to the application. It may be implemented as a low-level library and some other high-level libraries may be implemented on top of it.

When utilising parsers to parse DFDL descriptions, the whole process can be conceptualised in terms of 3 primary layers (see Figure 3 [5]). The central layer is the Abstract Data Model. In this layer, we focus on the primitive data structure such as integers, floating point numbers and strings, as well as some compound structures like vectors and arrays. Here we do not need to care about how many bits an integer or a double precision number occupies: we think of them as abstract entities. Their physical representations on the storage or in memory are hidden at this layer.

The lower layer is the mapping between abstract data structures and their physical representations, which defines, for instance, whether an integer is made up of 32 bits or 64 bits and whether it is represented as a flat data stream or a string. This determines how many bits are read and if transformations are required when extracting the figure. Similarly, we can define how individual elements in an array are separated from each other: by a space or a dot, etc. DFDL encapsulates the semantics of these basic definitions and transformations in an extensible set of primitive mappings that can be composed in sophisticated ways [5]. After defining the mappings in the lower layer, we associate abstract data models with their particular physical representations, and this is why conceptual data types in the central layer can be dealt with independently.

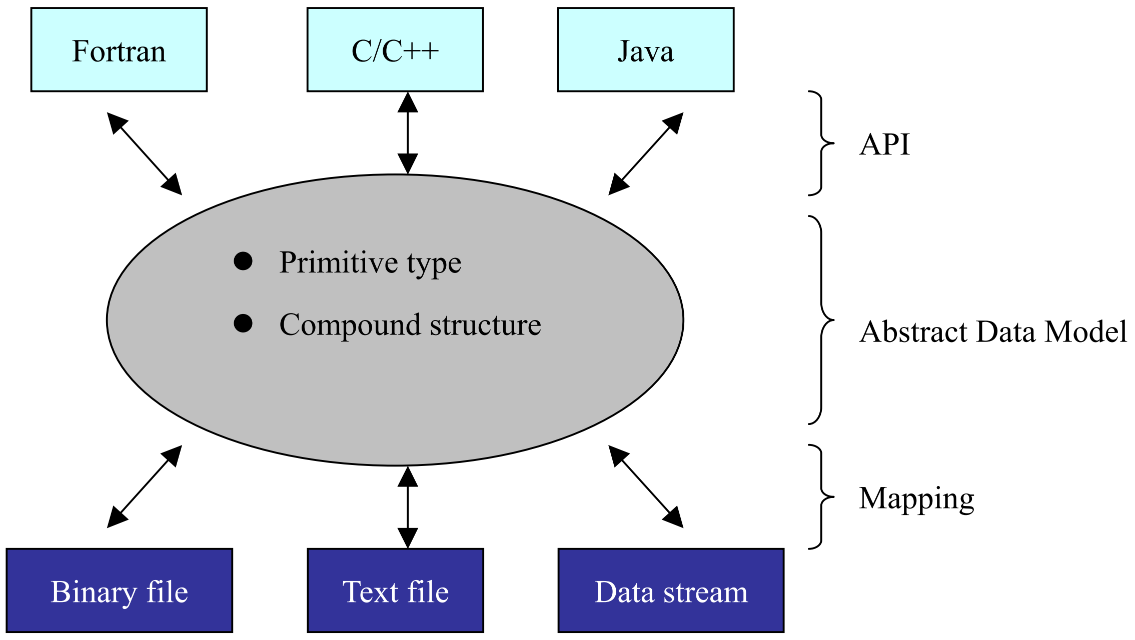


Figure 3 The architecture of DFDL

The upper layer, which can be thought of as a set of APIs, defines how an application written with a certain programming language accesses the actual data described by the abstract model along with the mapping. These APIs may have some additional sophisticated functions that enable instantiating values on demand and avoiding reading unnecessary bits, which are not part of the requested value, as well as caching.

All in all, applications written in a variety of programming languages use the upper level APIs to access the required data described by DFDL via central level abstract data models and lower level mappings. Sometimes the data is stored in a flat binary file: the first 4 bytes might definitely be the first integer; and sometimes the data is stored in a text file: the integer is represented by its ASCII value, so a transformation is required before we know what the value is. However, all of these are transparent for the high level applications. They only need to say, “I want the integer value”.

2.2 Simple examples

Thinking of the simple scenario mentioned in section 1.2, there are two heterogeneous data resources: a binary data file and a text file, and now we are going to demonstrate the working mechanism of DFDL by describing them. For a clearer understanding, we assume that both the binary file and the text file contain two consecutive values, an integer “1” and a float “1.0”. However, in the binary file, the byte sequence is

```
00 00 00 01 3F 80 00 00
```

In the text file, the byte sequence is

```
31 2C 31 2E 30
```

2.2.1 Describing the binary data file

According to the DFDL architecture, we need an abstract data type and a mapping which specifies a restriction on the raw data, determining how to access it. After defining a mapping between an ordinary XSD data type and its physical representation, we call it *a mapped type*.

Thus, in a similar process to the one described in [5], the first task is to define the mappings:

```
<dfdl:definitions>
   <dfdl:mapping name=”dfdl:data-bytes” rangeType=”dfdl:data”
      domainType=”dfdl:bytes.unbounded” direction=”bidirectional”/>
   <dfdl:mapping name=”dfdl:bytes-int” rangeType=”dfdl:bytes.4”
      domainType=”xs:int” direction=”bidirectional”/>
   <dfdl:mapping name=”dfdl:bytes-float” rangeType=”dfdl:bytes.4”
      domainType=”xs:float” direction=”bidirectional”/>
</dfdl:definitions>
```

Notice that we firstly mapped the raw data into bytes; and then we defined that an integer contains 4 bytes and so does a float. The attribute “name” specifies the name of the mapping; the “rangeType” and the “domainType” define the target data type and source data type respectively; and the “direction” attribute means whether the mapping is in single direction or bi-direction. With these mappings, next we need to define the mapped types which are the abstract types along with the mapping restrictions that they must follow.

```
<xs:simpleType name=”binaryInt”>
   <xs:restriction base=”xs:int”>
      <xs:annotation>
         <xs:appinfo>
            <dfdl:compositeMapping>
               <dfdl:mapping name=”data-bytes”/>
               <dfdl:mapping name=”bytes-int”/>
            </dfdl:compositeMapping>
         </xs:appinfo>
      </xs:annotation>
   </xs:restriction>
</xs:simpleType>


<xs:simpleType name=”binaryFloat”>
   <xs:restriction base=”xs:float”>
      <xs:annotation>
         <xs:appinfo>
            <dfdl:compositeMapping>
               <dfdl:mapping name=”data-bytes”/>
               <dfdl:mapping name=”bytes-float”/>
            </dfdl:compositeMapping>
         </xs:appinfo>
      </xs:annotation>
   </xs:restriction>
</xs:simpleType>
```

Now we have mappings and mapped types which can be thought of as bridges between the abstract data models and the mappings. Therefore we are ready to describe the binary data file using DFDL:

```
<xs:annotation>
  <xs:appinfo>
    <dfdl:definitions>
        <!—- global attributes -->
        <dfdl:dataFormat byteOrder=”bigEndian”/>
        <!-- mapped type assignments -->
        <use type=”dfdl:binaryInt”/>
        <use type=”dfdl:binaryFloat”/>
    </dfdl:definitions>
  </xs:appinfo>
</xs:annotation>

<xs:element name=”simpleExampleOfBinaryFile”>
  <xs:complexType>
    <xs:sequence>
        <xs:element name="x" type="xs:int"/>
        <xs:element name="y" type="xs:float"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Just as we said in Chapter 1, it is obvious that the entire DFDL description is made up of some XML Schema elements, such as *simpleType*, *complexType* and *sequence*, and some DFDL annotations which define mappings and supply attributes. For instance, the *byteOrder* attribute specifies an important characteristic for the mapping: the values conform to the *BigEndian* coding rule which means the most significant bytes are stored first.

Furthermore, we can see that the abstract data model and mapping are separated from each other. As in the central layer we only deal with abstract data types like integer and float, the binary file is defined as a sequence of an XSD integer and an XSD float. For associating abstract types with mappings, the “*use*” tag, which means “uses the mapped type”, is introduced. It places some restrictions on the ordinary *xs:int* and *xs:float*, implying that every time we meet a *xs:int* or a *xs:float*, it should be interpreted as a binaryInt or a binaryFloat. The advantage of separating abstract models from their mappings is that different mappings can be easily applied to the same abstract data model, and all low-level modifications of mappings are transparent to the high-level applications: even if the integer contains 8 bytes rather than previous 4 bytes, it is still a *xs:int* for the application.

According to this DFDL description, the original binary data file can be interpreted and then represented in the XML format as:

```
<simpleExampleOfBinaryFile>
   <x>00 00 00 01</x>
   <y>3F 80 00 00</y>
<simpleExampleOfBinaryFile>
```

2.2.2 Describing the text file

According to our assumption, the text file also consecutively contains an integer value and a float value, but stored in ASCII format instead of binary one. Again, beginning with the definition of mappings:

```
<dfdl:definitions>
   <dfdl:mapping name=”dfdl:data-string” rangeType=”dfdl:data”
      domainType=”xs:string” direction=”bidirectional”/>
   <dfdl:mapping name=”dfdl:string-int” rangeType=”dfdl:string”
      domainType=”xs:int” direction=”bidirectional”/>
   <dfdl:mapping name=”dfdl:string-float” rangeType=”dfdl:string”
      domainType=”xs:float” direction=”bidirectional”/>
</dfdl:definitions>
```

The idea is similar, but this time the mapping is via a string rather than bytes. Subsequently, the mapped types *textInt* and *textFloat* are defined in the same way.

```
<xs:simpleType name=”textInt”>
    <xs:restriction base=”xs:int”>
    <!-- using <xs:restriction base=”xs:float”> for textFloat -->
        <xs:annotation>
            <xs:appinfo>
                <dfdl:compositeMapping>
                    <dfdl:mapping name=”data-string”/>
                    <dfdl:mapping name=”string-int”/>
                    <!-- using <dfdl:mapping name=”string-float”>
                        for textFloat -->
                </dfdl:compositeMapping>
            </xs:appinfo>
        </xs:annotation>
    </xs:restriction>
</xs:simpleType>
```

Finally, the text file can be described as:

```
<xs:annotation>
  <xs:appinfo>
    <dfdl:definitions>
       <!-- attributes -->
       <dfdl:dataFormat characterSet=”UTF-8”/>
       <dfdl:dataFormat decimalSeparator=”.”/>
       <dfdl:dataFormat fieldSeparator=”,”/>
       <!-- mapped type assignments -->
       <use type=”dfdl:textInt”/>
       <use type=”dfdl:textFloat”/>
    </dfdl:definitions>
  </xs:appinfo>
</xs:annotation>

<xs:element name=”simpleExampleOfTextFile”>
  <xs:complexType>
    <xs:sequence>
       <xs:element name="x" type="xs:int"/>
       <xs:element name="y" type="xs:float"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Just as we discussed before, whether the values are physically represented as ASCII or binary bits is transparent for the high-level applications. Since the abstract data models for the text file are still an integer and a float, the sequence definition is the same. However, as the mappings are different, different mapped types are applied.

In addition, this time the attribute annotations are more complex and also more interesting. The first attribute tells us that the file uses the UTF-8 coding form for the ASCII characters; the second attribute defines that the float number uses "." as its decimal separator; and the final attribute specifies that individual strings are separated from each other via ",".

According to this DFDL description, the original text file can be interpreted and then represented in the XML format as:

```
<simpleExampleOfBinaryFile>
   <x>1</x>
   <y>1.0</y>
<simpleExampleOfBinaryFile>
```

2.3 Comparison with other file-oriented data formats

Besides DFDL, there are some other file-oriented data formats that can be used for data interchange in the future.

The HDF5, which is a completely new Hierarchical Data Format, consists of a data format specification and a supporting library implementation [9]. It defines a physical file format for particularly storing scientific data and generates self-describing binary data files containing complete information on their structure. However, DFDL aims to describe any type of data. Furthermore, HDF5 is prescriptive in that it is a format of data, and any HDF5 data files and applications must conform to this format. In contrast with it, DFDL is descriptive in that it is not a particular data format, but a way to define any data format. For instance, to generate a scientific data file using HDF5, it must be constructed based on the HDF5 primary objects: datasets and groups (see [9] for more details); while by using DFDL, the data file itself can be produced in any data format based on the requirement, and then a separate DFDL description file is written to describe this format.

Similar to DFDL, BinX is another means of describing the layout of binary data files (see [10] for more details). Both of them are descriptive and XML-based, sitting in a separate file from the original data. However, they are different in some small details, such as:

- BinX descriptions are real XML files and DFDL descriptions are XML Schema in substance, although they are all based on XML.

- One BinX description is associated with a particular data file or a group of data files that have the identical structure, while a DFDL description can be used to describe a set of related files whose structures are similar but not the same.

- The contents of every element and every property of BinX descriptions must be definite, however, XPath expressions can be used as variables for properties in the DFDL.

Chapter 3 JPEG data format

3.1 Digital Image concepts

Nowadays, JPEG, the acronym for “Joint Photographic Experts Group”, is a widely accepted international standard for both gray scaled and colourful images. It can be used almost anywhere: on the Internet, in scientific research and in peoples’ daily life. By definition, JPEG is concerned primarily with images that have two spatial dimensions, contain gray scale or colour information, and possess no temporal dependence [11], which is why we say that it is a standard for still images.

Usually, when we talk about JPEG, we mean “digital images” composed of a series of binary numbers that are converted from a sequence of sample points along each scan line at regular intervals. There are three very important concepts for digital images: sampling, quantisation and encoding.

Sampling is the first step of the process of converting continuous analog image information into discrete digital representations, by measuring the value of analog data at regular intervals; then splitting the continuous range of values into discrete levels and assigning sampling values to corresponding levels, which is called quantisation; finally representing those discrete levels by a particular set of binary numbers, which is known as encoding.

For instance, consider a one-dimensional analog image signal, shown in Figure 4. When we measured it at regular intervals, we were sampling. To make the fixed number of bits sufficient to represent the signal, we must discretize the decimal values, and in our example, divided the whole domain into 8 levels which can be represented by 3 binary bits. Subsequently, we used the simplest encoding to represent them. Finally, this continuous signal can be converted into digital representation as: 011 101 011 100 101 110 101 010 010 100.

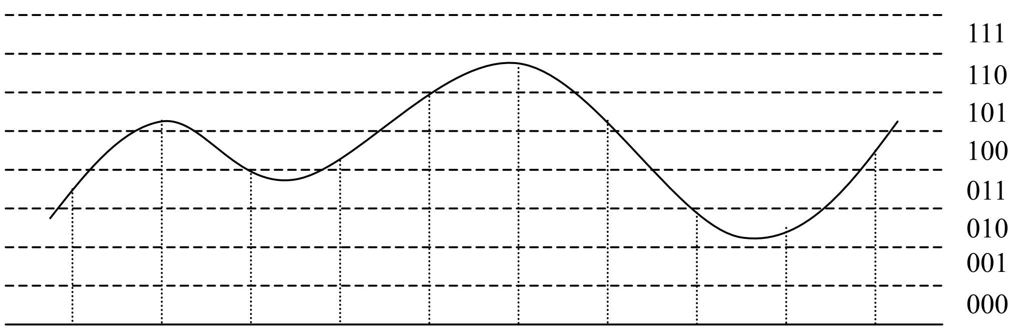


Figure 4 Sampling, quantisation and encoding illustration

There are three important parameters: precision, resolution and aspect ratio.

- The precision specifies how many bits are used for representing one sample point. The higher the precision is, the more levels of intensity can be represented, so the more details can be included [12]. In the previous example, the precision is 3 bits/sample.

- The interval between samples determines the resolution of the sampled signal. The smaller the sampling interval, the higher the resolution, and the more accurately we represent the continuous signal [12]. The resolution of our example is 10.

- The final parameter, aspect ratio, is particularly for 2-D images. It describes the shape of the sample, and this is determined by the relationship between the physical size of the image and the rectangular grid of samples [12].

3.2 JPEG file layout

One of the reasons why I chose JPEG as the data format that I want to describe using DFDL is that it has clear structures. JPEG files contain two classes of segments: entropy-coded segments which contain the entropy-coded image data, and marker segments which contain header information, tables, and other information required to interpret and decode the compressed image data [13]. As the “JPEG” in this project actually refers to JPEG JFIF as we mentioned in section 1.4, there are also some marker segments containing information like aspect ratio and orientation of the image, which are not needed to decompress the data, but are needed by many applications. Marker segments always begin with a “marker”, and all marker segments and entropy-coded segments are followed by another “marker”, so the entire data structure is clearly labeled. Figure 5 demonstrates the typical structure of a JPEG file.

```
SOI                                  Start of image
  APP0                               JFIF segment marker
  DQT                                Quantisation table
  definition
  DRI                                Restart interval
    SOFn                             Start of frame
      DHT                            Huffman table definition
      SOS                            Start of scan
        Entropy-coded segment, RST0
        … etc. …
        Entropy-coded segment, RSTn
      DHT                            Huffman table definition
      SOS                            Start of scan
        … etc. …
EOI                                  End of image
```

Figure 5 Structure of JPEG files

A marker is a unique two-byte code beginning with one-byte 0xFF and a non-zero one-byte marker code that specifies what the segment is. The SOI and EOI are the start and end of the image; APP0 is the JFIF segment marker: this segment provides attributes like the colour space and pixel aspect ratio. The markers for them are 0xFFD8, 0xFFD9 and 0xFFE0 respectively. In JPEG files, an image is usually represented in one or more frames, and SOF$_n$ (0xFFC0) is the start of frames. Each frame is further broken down into one or more scans, each of which appears as an entropy-coded bit stream, and SOS (0xFFDA) is the start of scans. Each frame and scan is preceded with a marker segment containing optional definitions for instance Quantisation tables[1] (DQT, 0xFFDB) and Huffman coding tables[2] (DHT, 0xFFC4) [6].

In addition to the entire file structure, some markers have their own structures. Amongst them, there are two very important markers, APP0 and $SOF_{n,}$, which are also of interest to my projects. The parameters of the APP0 marker and SOF_n marker are listed in Figure 6 and Figure 7 [8] [14]. As we can see, each marker has a well defined parameters list, and each parameter is allocated fixed number of bytes, which is crucial for the DFDL description. Subsequently, the parser will know the exact meaning of each bits block and the high-level library will extract the useful information, for instance, the width and height of the image.

[1] The set of 64 quantization values used to quantize the DCT coefficients.

[2] The set of variable length codes required in a Huffman encoder and Huffman decoder.

Parameter	Size	Description
Marker Identifier	2 bytes	0xFF, 0xE0
Length	2 bytes	This must be >= 16
File identifier mark	5 bytes	Using (0x4A, 0x46, 0x49, 0x46, 0x00) to identify JFIF
Major revision number	1 byte	This should be 1, otherwise error
Minor revision number	1 byte	This should be 0..2
Units for x/y densities	1 byte	0: no units, x/y-density specifies the aspect ratio instead; 1: x/y-density is dots/inch; 2: x/y-density is dots/cm
X-density	2 bytes	It should be not equal 0
Y-density	2 bytes	It should be not equal 0
Thumbnail width	1 byte	The width of thumbnail
Thumbnail height	1 byte	The height of thumbnail
Bytes to be read	n bytes	n = width * height * 3 bytes

Figure 6 Parameters of APP0 marker

Parameter	Size	Description
Marker Identifier	2 bytes	0xFF, 0xC0
Length	2 bytes	8 bytes + number of components * 3 bytes, excluding the 2 bytes allocated to the marker identifier
Sample precision	1 bytes	Bits/sample, usually 8
Number of lines (Image height)	2 bytes	This must be > 0
Number of samples per line (Image width)	2 bytes	This must be > 0
Number of components	1 bytes	Usually 1 = grey scaled, 3 = colourful
Specification for Each Component	3 bytes	Each component has 3 bytes, containing component id (1 byte), sampling factors (1 byte), and quantisation table number (1 byte)

Figure 7 Parameters of frame header

Chapter 4 Analysis

4.1 Requirements analysis

Requirements analysis is of great importance to any software development project. Without keeping the right requirements in mind, your project will definitely fail no matter how wonderful the design, coding and other activities are, as it will not produce the same thing as your customer expected. Furthermore, without a predetermined, detailed and legible requirements description, it would be extremely hard to track the entire project and to know whether your result would meet all your customer's expectations.

Although there is no tangible customer as part of my project, we can consider the DFDL working group as the potential customer who wants me to perform some exploration and experimental implementation based on a subset of DFDL and JPEG data format. Therefore, I need to know exactly what the DFDL-WG expected. So I discussed with experts of the DFDL-WG all over the world via E-mail and international phone call, in order to find out what they are really interested in.

Usually the requirements analysis covers two categories of requirements: functional requirements, which specify what activities the software must be capable of performing, and non-functional requirements, which describe how the software performs these activities. The outcome of requirements analysis is often a prioritised checklist of detailed requirements.

4.1.1 Functional requirements

Through discussion with the DFDL-WG, we found the most important functional requirement to be describing the JPEG format using DFDL by defining different markers segments. The second one is implementing a generic parser that can understand any DFDL description. Subsequently, a parser for parsing the original data file according to the information gained from the DFDL parser is to be developed. Finally, we need to implement a library, which produce an XML representation of the original data file utilising the two parsers. However, as we used the "Design to Schedule" development model (see next section for more details), the time remaining is a key factor for later requirements. The final checklist of functional requirements is shown in Figure 8. Each requirement is further broken down into a series of details with identifiers, so that we can examine each specific sub-requirement and refer to it in later chapters if required.

Priority	Requirement	Details
1	Describe the JPEG data format using DFDL	1. Define the SOI, EOI markers;
		2. Define the APP0 marker segment;
		3. Define the APP1 marker segment;
		4. Define the SOF0 marker segment;
		5. Define the SOS marker segment;
		6. Define the DHT marker segment;
		7. Define the DQT marker segment;
2	Develop a generic parser for parsing any DFDL description	8. Interpret different global attributes;
		9. Interpret simple types which is based on XML schema primitive types;
		10. Interpret complex types which comprise a series of simple and complex elements;
		11. Process simple type elements;
		12. Process complex type elements;
3	Develop a data file parser for parsing the data file according to the information gained from the previous DFDL parser;	13. Process different attributes;
		14. Process simple type data items;
		15. Process complex type data items;
4	Develop a library for representing the original data file in XML format	16. Represent the original data file in XML format;

Figure 8 The prioritised checklist of functional requirements

4.1.2 Non-functional requirements

With functional requirements, we specify what the software should do, while with non-functional requirements, we want to describe how it does it. As listed in Figure 9, the most important requirement is "effectiveness", which means that the parser must be able to effectively perform right actions on a wide range of data files besides JPEG images, otherwise we cannot say that the software is successful. Another important requirement is "portability", because DFDL is part of the data management technologies for the Grid, which mainly focuses on the integration of heterogeneous data resources on different platforms. The other three requirements concern the quality of the software, but sometimes these are difficult to examine and address.

Priority	Requirement	Details
1	Effectiveness	1. Working correctly with JPEG images;
		2. Working correctly with binary data files;
		3. Working correctly with text files;
2	Portability	4. Being able to work on Windows platform;
		5. Being able to work on Sun Solaris platform;
		6. Being able to work on Redhat Fedora Linux platform;
3	Efficiency	7. Reasonable performance on different kinds of data files, and files with different size;
3	Scalability	8. Working correctly with simple files (Less than 10 MB);
		9. Working correctly with big files (Around 500 MB);
3	Usability	10. Being able to be easily used by clients.

Figure 9 The prioritised checklist of non-functional requirements

After finishing the requirements analysis and producing the prioritised checklists, what we must do is to follow these requirements and mark each one after achieving it. They exist for the purpose of helping us understand what we need to achieve, and to highlight our developing direction.

4.1.3 High-level generalisation

The Use Case Diagram is a powerful tool to generalise the high-level requirements of a system and demonstrate the interaction between an actor, who is the image user in our case, and the system. Through performing this interaction, the actor can accomplish certain goals. It is a concise and explicit means of capturing and documenting a high-level view of what the system does, so I use it to specify and visualise the requirements from a more general perspective.

From this point of view, the purpose of the system at the highest-level is to help the image user convert the original JPEG image file into an XML representation, which is portable and understandable to almost all platforms, as shown in Figure 10.

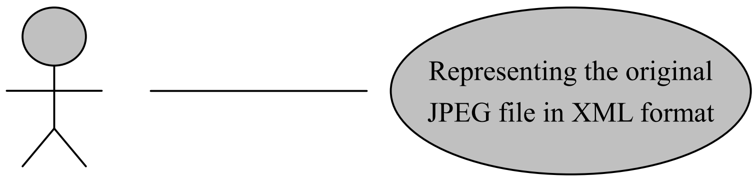


Figure 10 Highest-level of the Use Case Diagram

For showing more details, we draw the second-level Use Case Diagram: breaking down the highest-level requirement into two sequential sub-requirements, see Figure 11.

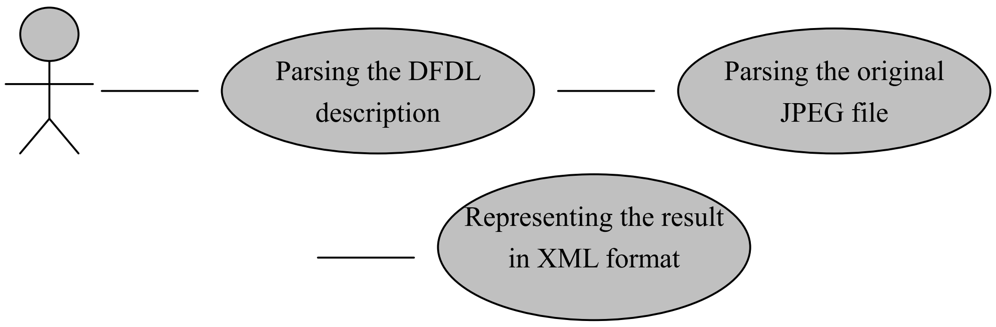

Parsing the DFDL description
Parsing the original JPEG file
Representing the result in XML format

Figure 11 Second-level of the Use Case Diagram

4.2 Development model & process

As we know, there are many different development models, such as waterfall, evolutionary delivery, and spiral model, which are suitable for different kinds of problems. Amongst them I chose to use "Design to Schedule" because:

- Firstly, it is an iterative method, which would give me a tangible result at an early stage, show me an instant view where the project has arrived, and make myself confident that it is progressing in the right direction.

- Secondly, this project is only a subset of the DFDL project, without complicated requirements and stages.

- Thirdly, we have a hard deadline, but the requirements are relatively flexible as this project is somewhat heuristic and tentative: it does not mean that there are no specific requirements, but we can decide what to implement; if there is enough time, I can add some more features to implement; if no time is available, we can stop.

In detail, the first step of Design to Schedule, which is also the most important one and will determine the success of the model, is to order features in terms of importance. The result of ordered features is similar to the prioritised functional requirements shown in Figure 8 and Figure 9. Secondly, the most critical features are implemented in the earliest releases, with less important features implemented if there is time and stopping when time runs out.

In addition to the development model, the development process is also of great significance for the success of the project. It is an implementation of sensible activities to keep a project on track, for instance, revision control, testing and risk management. Revision control is simply archival and retrieval of specific working material, or in other words, maintaining a history of source files and records of changes over time in a retrievable way. We will discuss the testing in detail in section 6.4, so we only talk about risk management here.

Risk management is the area of process which identifies and handles project risks, including risk assessment and risk control. After risk assessment, we will get a list containing the description of the risk, the chance of it occurring, its impact on the schedule and more important, the priority extrapolated from the previous two factors, as well as a reasonably proposed solution. The risk list of our project is shown in Figure 12, and the unit for "impact" is how many "weeks" the risk will affect the entire project, i.e. if we assume the first risk, the expected overrun would be six weeks.

Risk	Chance of occurring	Impact (weeks)	Priority	Solution
Not familiar with DFDL which is absolutely new	90%	6	5.4	Discussing with DFDL working group using mailing list or telephone;
Inexperienced in the formal software engineering principles	70%	5	3.5	Communicating with my supervisors as often as possible.
Maybe the existing DFDL specification does not support some features which we want such as variable length bits block	50%	6	3	Discussing with DFDL working group to find a suitable solution, or choosing other features to implement
Do not have enough mathematics background which is required by fully understanding the JPEG	70%	2	1.4	Asking my friends who are competent in mathematics or signal processing for help.

Figure 12 Prioritised list of risks and corresponding solutions

However, risk assessment is not our final objective, instead we intend to draw support from it and keep better control of these risks. The more accurately risks are identified before hand, the earlier we will solve them, and accordingly the bigger chance for success we will gain.

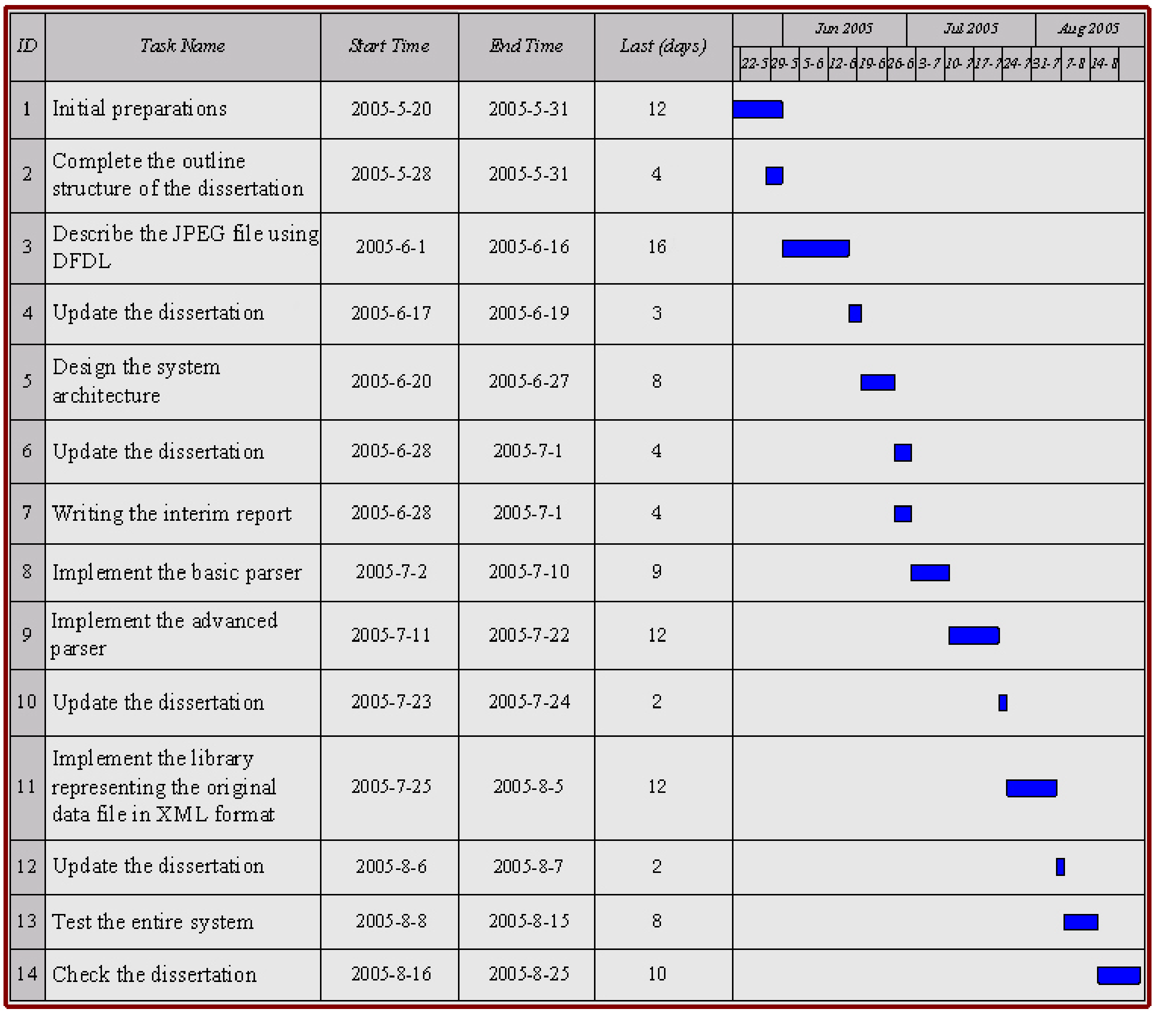
4.3 Work plan

The Gantt chart for the entire project is shown in Figure 13. I divided my project into 14 tasks and 3 phases. The first phase is from the end of May to July, I will understand, analyse and design the entire project, as well as updating the dissertation and finishing the interim report. The phase 2 is from July to early August, implementing the parser and the libraries. The phase 3 is testing phase. I will complete it 10 days before the deadline and then checking the dissertation. I did not think that leaving the dissertation to the last couple of weeks was a good idea, so I began the dissertation in June, and continued to update it several times during the 3 months, leaving enough time for checking at the end.

4.4 Development model in practice

The entire project followed the “Design to Schedule” development model very well. I wrote the DFDL description for the JPEG data format, which has the highest priority, in the first stage, and then implemented the generic DFDL parser and the data file parser one after the other, and finally developed a library method for representing the original data file in XML format. This development model provided me with tangible results at the earliest stages, which made me confident that the project was going well.

The project also benefited from the “risk management” development process. At the first stage, there was a misconception of generating DFDL descriptions on a one-to-one basis with different JPEG files. However, since we had already identified the risk that we were not very familiar with DFDL, we discussed with the DFDL working group and changed our work plan in time. The original work plan put more emphasis on the details of each JPEG image, but the new plan concentrated on using and parsing the DFDL itself. As we corrected our decision in the earliest stage, it did not seriously affect the entire project.



ID	Task Name	Start Time	End Time	Last (days)
1	Initial preparations	2005-5-20	2005-5-31	12
2	Complete the outline structure of the dissertation	2005-5-28	2005-5-31	4
3	Describe the JPEG file using DFDL	2005-6-1	2005-6-16	16
4	Update the dissertation	2005-6-17	2005-6-19	3
5	Design the system architecture	2005-6-20	2005-6-27	8
6	Update the dissertation	2005-6-28	2005-7-1	4
7	Writing the interim report	2005-6-28	2005-7-1	4
8	Implement the basic parser	2005-7-2	2005-7-10	9
9	Implement the advanced parser	2005-7-11	2005-7-22	12
10	Update the dissertation	2005-7-23	2005-7-24	2
11	Implement the library representing the original data file in XML format	2005-7-25	2005-8-5	12
12	Update the dissertation	2005-8-6	2005-8-7	2
13	Test the entire system	2005-8-8	2005-8-15	8
14	Check the dissertation	2005-8-16	2005-8-25	10

Figure 13 Gantt chart for the entire project

In addition, the timetable for dissertation deliveries is demonstrated in Figure 14.

Time	**Deliverables**
1st June – 5th June	Chapter 1 Introduction
6th June - 12th June	Chapter 2 DFDL
13th June - 19th June	Chapter 3 JPEG
20th June - 26th June	Chapter 4 Analysis
1st July	Interim report
1st July – 10th August	Implementing a little, updating a little (Chapter 5, 6, 7) (Updating every fortnight)
11th August – 16th August	Conclusion, integrating the entire article
17th August – 25th August	Checking

Figure 14 Time table for dissertation deliveries

Chapter 5 Design

5.1 Functionality design

According to my functional requirements analysis, the system should have the functionalities below:

(1) Reading the original JPEG data file;

(2) Reading the corresponding DFDL description;

(3) Parsing the DFDL description and understanding global attributes defined in the beginning of the description, such as the *byteOrder*;

(4) Parsing the DFDL description and understanding how to identify individual segments of the original data file: knowing how many bytes each segment occupies; what type of data they are representing; what the restrictions on their values are; what the threshold address of each segment is;

(5) Based on the structure information, extracting each segment of data;

(6) Representing those segments of data in XML format with the tags expressing what the data means.

5.2 System design

Now I am going to discuss the high-level design of the system from top to bottom, beginning with the *DFDLLibrary*, which is the top-level class of the system, and other core implementations in the following. The UML Class Diagram, which is a powerful and widely accepted tool for describing the structure of classes in a system, is used. At the end, an overview of the system design will be given, utilising the Class Diagram to show the classes structures and utilising the Sequence Diagram to illustrate their interactions.

5.2.1 DFDLLibrary

The functionalities described in 5.1 are encapsulated into a high-level library called *DFDLLibrary*, which provides a user with all the required interfaces. The encapsulation groups those related functionalities into one unit, which can thereafter be referred to by a single name. It provides a series of interfaces: parsing the DFDL description of JPEG format as well as the image file; retrieving the structure described in the DFDL description and the actual data; representing the data in XML format. A UML Class Diagram, which is a powerful and widely accepted tool for describing the structure of classes in a system, is used to visualise the *DFDLLibrary* (see Figure 15).

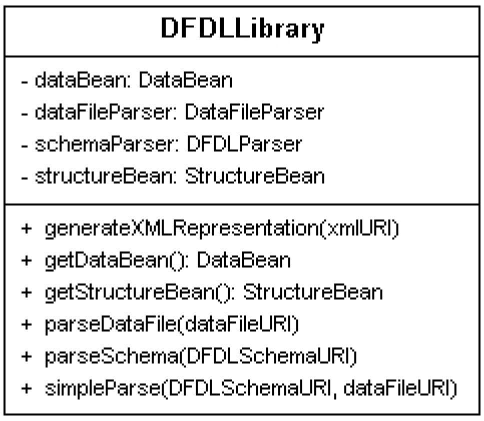

DFDLLibrary
- dataBean: DataBean
- dataFileParser: DataFileParser
- schemaParser: DFDLParser
- structureBean: StructureBean
+ generateXMLRepresentation(xmlURI)
+ getDataBean(): DataBean
+ getStructureBean(): StructureBean
+ parseDataFile(dataFileURI)
+ parseSchema(DFDLSchemaURI)
+ simpleParse(DFDLSchemaURI, dataFileURI)

Figure 15 Class Diagram of *DFDLLibrary*

Several benefits are gained from this approach. A user is not concerned with how the functionalities are implemented, but needs to be able to use them as simply as possible. Through encapsulation, implementation details are kept hidden from the user, thus providing the user with maximum convenience. For example, what the user usually wants to do is to parse a JPEG file according to the DFDL description, after the encapsulation he only needs to call the *parse()* method provided by the *DFDLLibrary* and specify the locations of the image file and the DFDL description, without the requirement of knowing how the process actually performs. Similarly, simply calling the *generateXMLRepresentation()* method will produce the XML representation of the original data file for the user. In fact, the method contains a sequence of activities, but the user does not need to know them. Encapsulating functionalities into *DFDLLibrary* makes the achievement of requirements much more straightforward.

In addition, the encapsulation helps to ensure that the system works correctly by hiding not only the implementation but also the data, preventing the user from changing them improperly. It should always be kept in mind that the user might not be a programming expert. Furthermore, the encapsulation is a means of increasing the usability of the system, which is one of our non-functional requirements. It makes the code reusable in other systems.

5.2.2 Beans

The official definition for a Bean is that "A bean is a reusable software component based on Sun's JavaBeans specification that can be manipulated visually in a builder tool." [15] Here I extended the concept of Bean to invisible components which have a series of closely related attributes and methods, and are reusable. Taking the student information as a simple example: a group of information, such as student number, student name, gender, age, and address etc., can be encapsulated together as a single unit - a student information bean, and then some *setter* and *getter* methods can be defined for setting and getting the information. These related data are thus grouped together and the object can be reused by different components of the system, additionally the same structure can be directly reused for different students. This will increase the readability and maintainability as well as the usability of the code.

In our system, there are two categories of information which are of great importance: the structure information of the JPEG data format and the actual data which have been segmented according to the structure information. Because they are critical intermediates of the whole system and they are required to be reused, I decided to introduce two beans encapsulating them respectively: one is the *StructureBean*, and the other one is the *DataBean*. Within each bean, the core is a tree data model which will be used to store the information from the DFDL description and the actual data. The tree data structure is used because the substance of a DFDL description is an XML schema, which is formatted with a tree-like structure; and the actual data should be organized with the same tree-like structure as the DFDL description. The tree data structure I would use is provided by the Data Structures Library in Java (JDSL), which is being developed at the Center for Geometric Computing, Department of Computer Science, Brown University [16]. I do not intend to develop a tree structure of my own, since it is a well-researched topic and its implementation would be time-consuming, however, the most important reason is that how to implement it is not the concern of my project and I do not want it to distract me from concentrating on DFDL.

The differences between these two beans are that: firstly, besides the structure information itself, the *StructureBean* should also store some global DFDL properties and type attributes that are necessary for understanding the structure; secondly each leaf node of the tree in the *StructureBean* is a composite type containing a sequence of structure information, which I call a *StructureItem*; whereas each leaf node of the tree in the *DataBean* is the actual data. A Class Diagram containing the *StructureBean* and *DataBean* is shown in Figure 16.

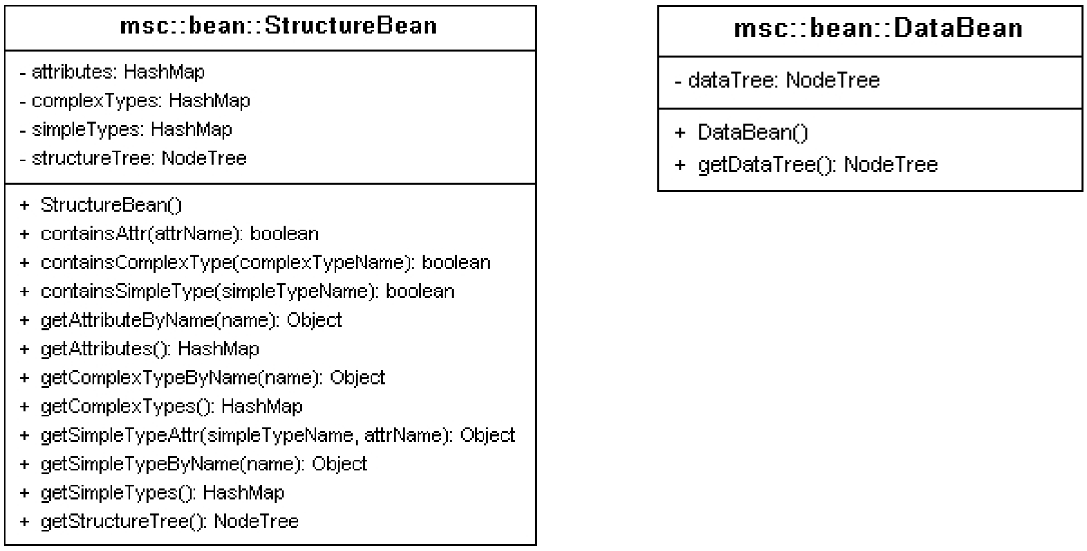

msc::bean::StructureBean
- attributes: HashMap
- complexTypes: HashMap
- simpleTypes: HashMap
- structureTree: NodeTree
+ StructureBean()
+ containsAttr(attrName): boolean
+ containsComplexType(complexTypeName): boolean
+ containsSimpleType(simpleTypeName): boolean
+ getAttributeByName(name): Object
+ getAttributes(): HashMap
+ getComplexTypeByName(name): Object
+ getComplexTypes(): HashMap
+ getSimpleTypeAttr(simpleTypeName, attrName): Object
+ getSimpleTypeByName(name): Object
+ getSimpleTypes(): HashMap
+ getStructureTree(): NodeTree
msc::bean::DataBean
- dataTree: NodeTree
+ DataBean()
+ getDataTree(): NodeTree

Figure 16 Class Diagrams of the *StructureBean* ***and*** *DataBean*

5.2.3 Parser

The *StructureBean* and the *DataBean* discussed in the previous section are the key data components of the system; in this section we are going to discuss the key processing components of the system: the *DFDLParser* and the *DataFileParser*. The first one is for parsing the DFDL description, resulting in a *StructureBean*; and the other one is for parsing the actual data file according to the *StructureBean* from the first step.

I have separated them into two steps because it is a common use care that a batch of pictures need to be parsed based on one DFDL description. The separation of the two parsers enables us to only parse the DFDL description once and then parse the images one by one. Otherwise the two parsing actions would happen at the same time, so both of them must be performed together several times, which leads to unnecessary overhead. Another reason for separating them is that performing them together makes the code much more obscure and much more complex, whereas keeping them separated makes the code much more legible, increasing the readability and maintainability.

However, the *DFDLParser* and the *DataFileParser* are not concrete classes, but interfaces. At the concept level, an interface defines a series of public member methods and no implementation, only specifying the APIs for certain functionalities; while at the implementation level, an interface cannot be instantiated but a concrete class can. Thus other two classes, the *SimpleDFDLParser* and the *SimpleDataFileParser*, which implements the two interfaces respectively, are defined. They provide the actual implementations of all methods declared in the interfaces. The most significant reason for separating the interface and the implementation is that it enables us to perform effective unit testing – designing the test classes before writing the code – ensuring different interfaces are correctly invoked.

The Class Diagrams of the two interfaces and their implementations are shown in Figure 17.

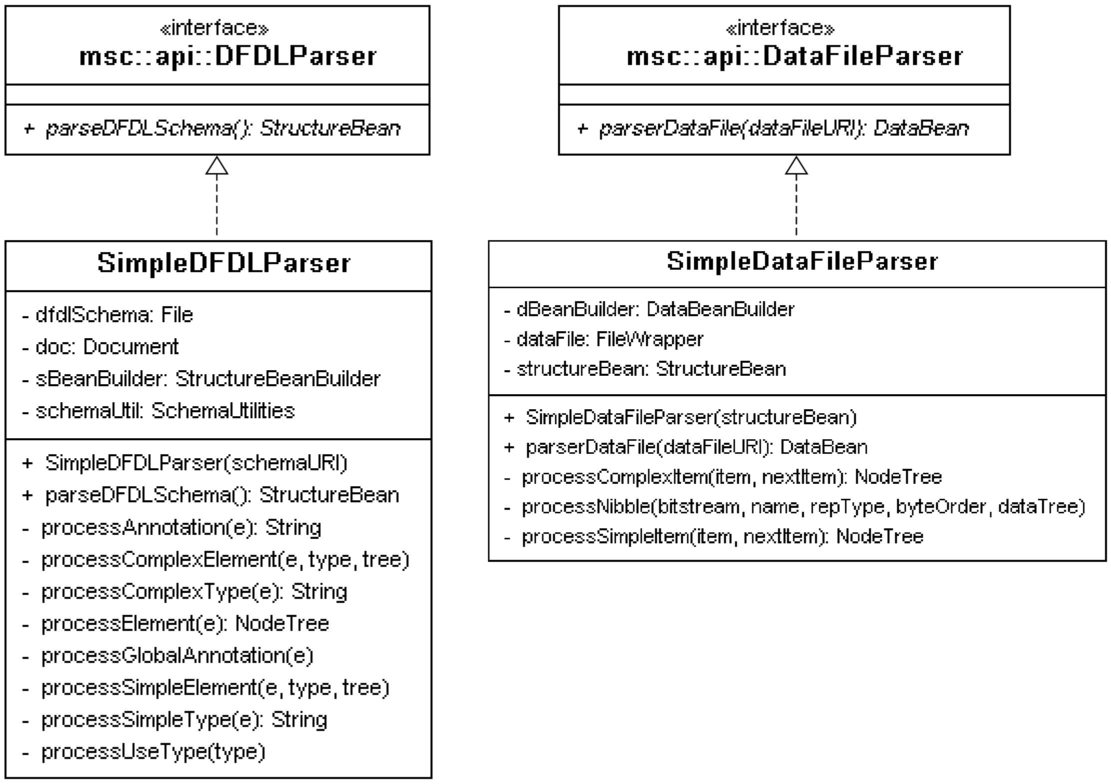

«interface»
msc::api::DFDLParser
+ parseDFDLSchema(): StructureBean
«interface»
msc::api::DataFileParser
+ parserDataFile(dataFileURI): DataBean
SimpleDFDLParser
- dfdlSchema: File
- doc: Document
- sBeanBuilder: StructureBeanBuilder
- schemaUtil: SchemaUtilities
+ SimpleDFDLParser(schemaURI)
+ parseDFDLSchema(): StructureBean
- processAnnotation(e): String
- processComplexElement(e, type, tree)
- processComplexType(e): String
- processElement(e): NodeTree
- processGlobalAnnotation(e)
- processSimpleElement(e, type, tree)
- processSimpleType(e): String
- processUseType(type)
SimpleDataFileParser
- dBeanBuilder: DataBeanBuilder
- dataFile: FileWrapper
- structureBean: StructureBean
+ SimpleDataFileParser(structureBean)
+ parserDataFile(dataFileURI): DataBean
- processComplexItem(item, nextItem): NodeTree
- processNibble(bitstream, name, repType, byteOrder, dataTree)
- processSimpleItem(item, nextItem): NodeTree

Figure 17 Class Diagrams of the parsers

5.2.4 BeanBuilder

We have defined a high-level DFDLLibrary that provides users with simple APIs like *parse()*, two parsers which will be used by the *parse()* for actually parsing the DFDL description and the data file respectively, and two beans storing the parsing results. Now our concern is how the two parsers produce the two beans. The simplest and most straightforward method is to parse the file whilst at the same time writing information into the bean directly. However, in that case there would be strong coupling between the parser and the bean, which has high impact on the readability and the maintainability of the code.

Therefore, two *BeanBuilder*s based on the "Builder" design pattern are introduced. A builder is for the creation of object, with emphasis on encapsulating the construction processes and separating the construction of a complex object from its representation so that the same construction process can create different representations [17]. By using the *StructureBeanBuilder*, the *DFDLParser* does not interact with the *StructureBean* any more. Whenever information needs to be added to the bean, the parser calls the builder to perform appropriate construction actions; and finally the builder directly returns the resulted bean to the *DFDLLibrary*. The same happens with the *DataBeanBuilder*. The Class Diagrams of the *StructureBeanBuilder* and *DataBeanBuilder* is shown in Figure 18.

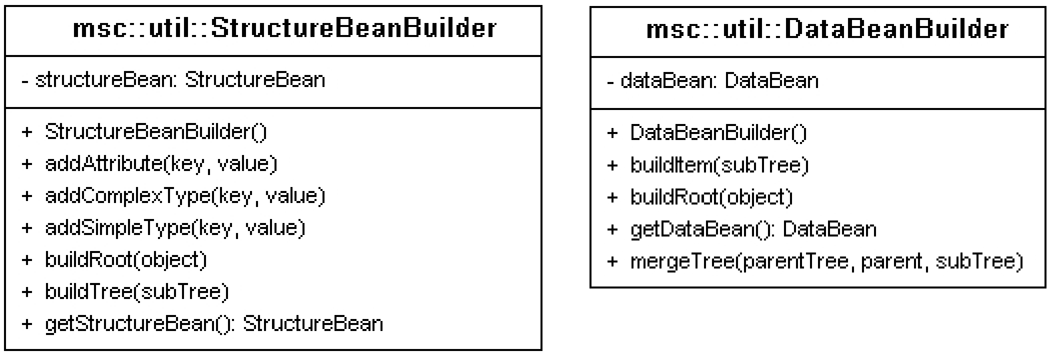

msc::util::StructureBeanBuilder
- structureBean: StructureBean
+ StructureBeanBuilder()
+ addAttribute(key, value)
+ addComplexType(key, value)
+ addSimpleType(key, value)
+ buildRoot(object)
+ buildTree(subTree)
+ getStructureBean(): StructureBean
msc::util::DataBeanBuilder
- dataBean: DataBean
+ DataBeanBuilder()
+ buildItem(subTree)
+ buildRoot(object)
+ getDataBean(): DataBean
+ mergeTree(parentTree, parent, subTree)

Figure 18 Class Diagrams of the BeanBuilders

5.2.5 System design overview

The main Class Diagram and Sequence Diagram of the entire system is shown in Figure 19 and Figure 20.

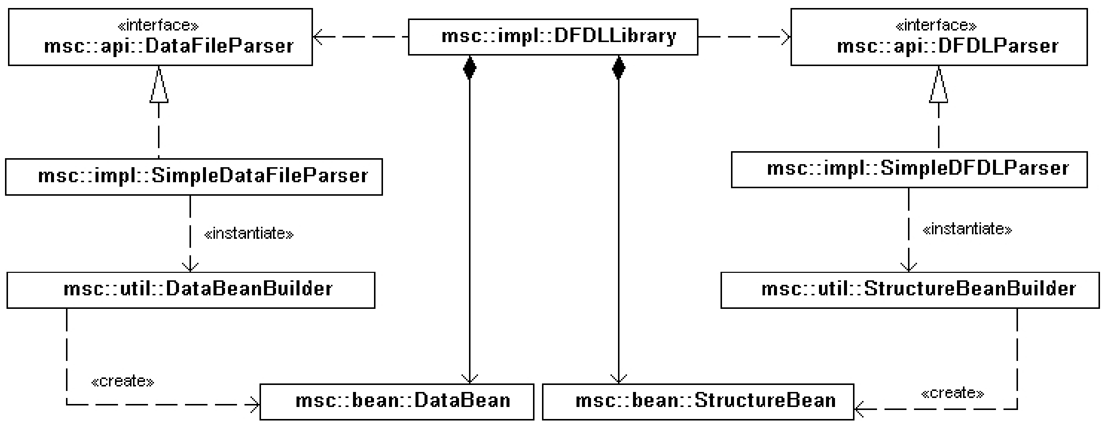


Figure 19 Class Diagram of the entire system

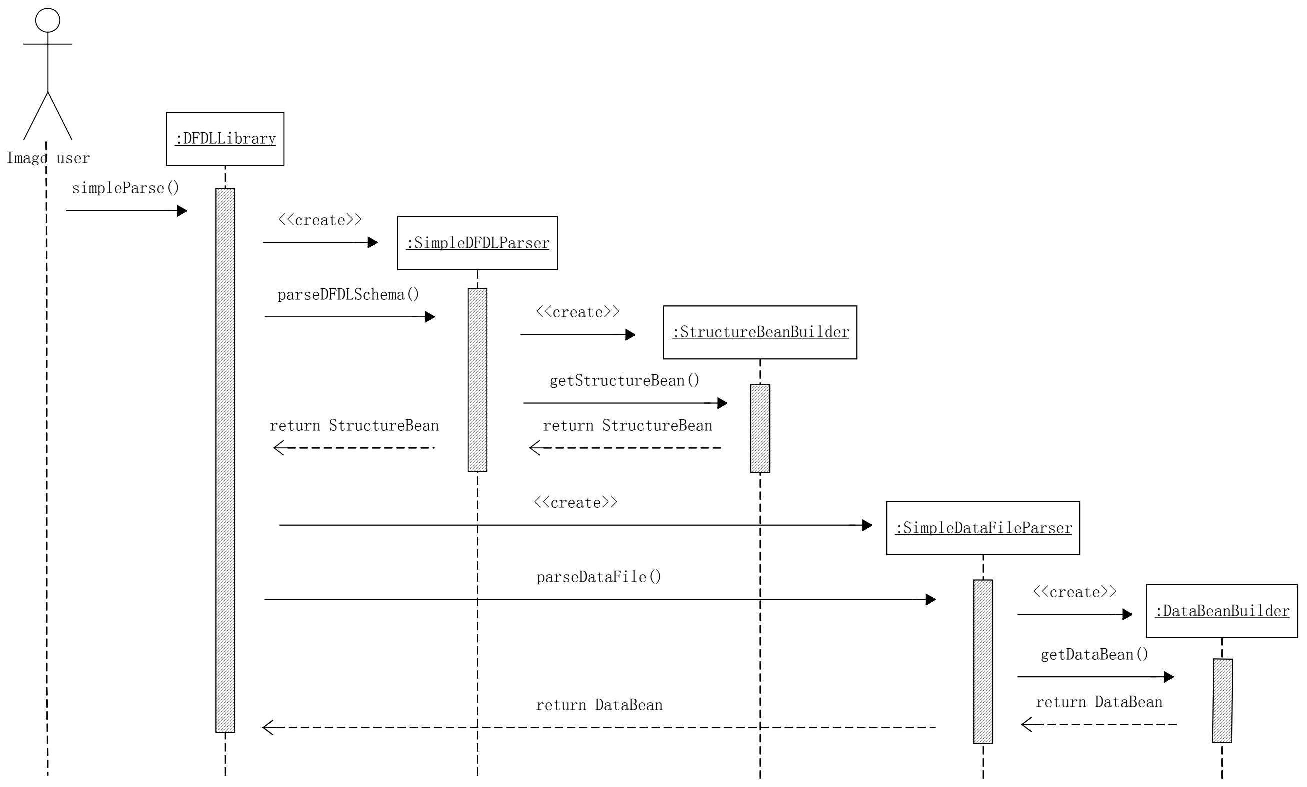


Figure 20 The sequence diagram of the system

Chapter 6 Implementation highlights

In the first section of this chapter, I will define the subset of DFDL that my implementation is based on, discuss how to implement the key points of the JPEG DFDL description, and spotlight how to solve the two most challenging problems of defining JPEG using DFDL. Afterwards I will explain the implementation of the DFDL Schema parser and the data file parser in sections 6.2 and 6.3 respectively. Rather than covering all the implementation details, I chose to put emphasis upon the significant pieces of work in the different parts of the implementation.

6.1 DFDL Schema

In section 2.2, two simple but typical DFDL examples are given, using the classical definition style, which is the best way to clarify the 3-level architecture and the core concept of DFDL: separating the abstract data models and their physical representations. However, after discussion with the DFDL working group, they suggested that it would be more practical and more useful to implement DFDL in an up to date simplified definition style, which will be described in section 6.1.1. Furthermore, since the DFDL specification itself is still under research and debate, it is impossible to cover all of its details, which are often changing, in the project. Therefore, I chose to implement a core subset of the DFDL specification that is less likely to change. The supported syntax and directives of DFDL Schema are defined in sections 6.1.2 and 6.1.3. After that, I will explain how the two particularly challenging difficulties – nibble elements and variable length segments – were solved.

6.1.1 The simplified definition style

In the classical definition style used in the previous examples, the mappings are defined explicitly and are separated from the mapped type definition that must specify what mappings it uses. This exactly coincides with the “API”, “Abstract Data Model” and “Mapping”, three layers of DFDL architecture. However, the expressions are a little tedious and difficult to parse. Consequently, as described in [18], a simplified definition method is introduced: combining the mapping and the mapped type definition together, using XML Schema attributes instead of separate definitions. For the same example – a binary data file containing an integer and a float – the mapped types can then be described as:

```
<xs:simpleType name=″dfdl:binaryInt″ dfdl:byteSize="4"
   dfdl:repType="binary">
   <xs:restriction base="xs:int"/>
</xs:simpleType>

<xs:simpleType name=″dfdl:binaryFloat″ dfdl:byteSize="4"
   dfdl:repType="binary">
   <xs:restriction base="xs:float"/>
</xs:simpleType>
```

which is also preceded by the “use” directives and followed by the same element definitions. Here we do not refer to any mapping explicitly, however all necessary information which is required to identify different data segments is clearly represented, including the type name, how many bytes it occupies, whether it is representing binary data or text data, and which abstract data type it is based on. For example, from the first definition we know that the mapped type *dfdl:binaryInt* is a binary data, based on the XML Schema primitive type *int* and physically occupying 4 bytes. Moreover, this way of definition can be simply extended by adding other attribute parameters, for instance, if the byte order of a particular data item is different from others, we can add an attribute *dfdl:byteOrder=“bigEndian”* which will override the global definition of byte order.

6.1.2 Overall syntax of DFDL Schema

Since the DFDL description is XML Schema based, it must generally conform to the XSD specification: beginning with a schema header, followed by a sequence of simple types definitions, complex types definitions and the actual elements that are of certain types. A simple type must base on a particular XML Schema primitive type or a DFDL type that has been defined. A complex type comprises a sequence of simple type or complex type elements. Both simple types and complex types can be defined either independently or implicitly within a particular element. An element must have a "name" attribute, and either a "*type*" attribute specifying what type it is or a new type definition immediately enclosed within the element. The occurrence of elements might be constrained by using "*minOccurs*" and "*maxOccurs*" attributes, which are "1" by default [19].

In ordinary XML Schema, it is permissible to have several elements which may appear in any order, but this is not allowed in DFDL Schema, because every data element in DFDL Schema must have a restriction on its occurrence order, as otherwise it is impossible to identify individual elements. Therefore if there are several elements, they must be assembled into a complex type element which places a restriction on the order of its children.

Besides ordinary XML Schema directives, there might be some DFDL directives embedded in a pair of "*<xs:annotation><xs:appinfo>*" and "*</xs:appinfo></xs: annotation>*" tags, which are called DFDL annotations. All DFDL related information is defined in this way. DFDL annotations may appear in two kinds of places: immediately after the XML Schema header or within an "*<xs:element>*" definition, which is the scoping issue of DFDL: if a DFDL directive appears at the beginning and between "*<xs:definitions>*" and "*</xs:definitions>*", it is a definition with global scope, which provides default attributes to all elements; if it appears within a particular element, it is a definition with local scope, which will override the default global attributes but only affect this element.

To clearly distinguish DFDL annotation directives and XML Schema directives, qualified element directives and type references must be used, which means the element tags and type names must be explicitly prefixed by the appropriate namespace. If the attributes are under the same namespace as the element they belong to, they can be unqualified, otherwise they also need to be qualified. For instance, the element tags "*xs:simpleType*", "*xs:annotation*", "*dfdl:dataFormat*", and type names "*xs:int*", "*dfdl:binaryInt*" must have the "*xs:*" and "*dfdl:*" prefix, specifying whether they come from XML Schema or DFDL. However, within an element like "*xs:simpleType*", any XML Schema attributes like "*name*" don't need prefix, but DFDL attributes like "*dfdl:byteSize*" still need prefix as they are under different namespace.

6.1.3 Basic DFDL directives supported

➢ "dfdl:dataFormat"

The "*dfdl:dataFormat*" directive is the most useful DFDL directive, specifying different kinds of data format properties. Its syntax is:

```
   <dfdl:dataFormat  property1=″value1″  property2=″value2″  …  />
```

With this directive, a wide range of mapping between abstract data type and physical representation can be specified using supported properties, including *repType*, *byteOrder*, *byteSize* and *terminator*.

The *repType* property specifies whether the data item represents a binary value or a text string, so it has two possible values: "*binary*" and "*text*". The *byteOrder* property specifies how multi-byte data are physically constructed, "*bigEndian*" or “*littleEndian*”. For instance, the actual value of the two bytes of data 0x0001 is 1 if the *byteOrder* is “*bigEndian*”, but it becomes 256 if the *byteOrder* is “*littleEndian*”. The *byteSize* property defines how many physical bytes the data item occupies, it might be either an integer (>=0), a mathematic expression or “*unbounded*”. The *terminator* attribute is useful when the *byteSize* of an element is “*unbounded*”, because an unbounded data item is ambiguous in the physical level, so it must be defined what data should be read to construct this data item. The value of “*terminator*” is a sting bracketed by “[” “]”, containing all candidate terminators separated by “,”.

- “dfdl:use”

The “*dfdl:use*” directive is used to integrate DFDL types with XML Schema primitive types. After a new simple DFDL type is defined, it can be linked with the primitive type that it is based on, indicating that whenever the primitive type is used, it should be interpreted that the DFDL type is used. For example, by using the following directive:

```
<dfdl:use type="dfdl:binaryInt"/>
```

implies that each *xs:int* element should be considered as a *dfdl:binaryInt* element, and processed on top of the structure information defined in the *dfdl:binaryInt* type. However, it is required that there must be an explicit declaration of this DFDL type, for instance:

```
<xs:simpleType name=″dfdl:binaryInt″ Property1="value1"
   Property2="value2">
   <xs:restriction base="xs:int"/>
</xs:simpleType>
```

- “dfdl:definitions”

Considering the usage of the “*dfdl:use*” directive, it should be a global definition.

However, how is a global definition expressed in DFDL? The answer is using "*dfdl:definitions*" directive. A group of attributes with global scoping should be declared by

```
<dfdl:definitions>
   Individual directives
</dfdl:definitions>
```

where the individual directives may include *repType*, *byteOrder*, etc.

Definitions provide a set of default value for different attributes, which will be propagated to all schema elements. However, if a simple type or an element has the corresponding attributes declaration of its own, these local definitions will override the global ones. For example, we can use

```
<dfdl:definitions>
   <dfdl:dataFormat byteOrder="bigEndian"/>
</dfdl:definitions>
```

to specify the global byte order attribute, and then we do not need to declare it every time except in the IFD0 element definition, where the byte order becomes littleEndian. Thus, we have to use local DFDL annotation to override the global value:

```
<xs:element name="numberOfEntry" type="xs:short">
   <xs:annotation>
      <xs:appinfo>
         <dfdl:dataFormat byteOrder="littleEndian"/>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

and it only affects this element.

6.1.4 Nibble elements

The definitions of nibble elements which consist of several bits rather than whole bytes is a challenge for describing JPEG using DFDL, since whether or not DFDL should support the definition of bit construction and how to define it are still under discussion. However, for some fields, like Internet protocol, data communication and JPEG, which concern low-level details of data, it is necessary to be able to define bit segments.

The central idea of defining bit streams is also to separate abstract data types and their physical representations. For a high-level API or application, bit streams can still be considered as bytes or even integers; however, their actual values come from the value of that several particular bits, and the other bits are filled with “0”. To distinguish the “nibble” definition from the ordinary byte stream definition, a new DFDL directive is introduced:

```
<dfdl:bitstream length=”value”/>
```

Where the tag name “*dfdl:bitstream*” indicates that this element is made up of several bits rather than ordinary whole bytes, and the number of effective bits is constrained by the “*length*” attribute.

For example, in JPEG format, there are two consecutive property segments, the "*horizontalSamplingFactors*" and the "*verticalSamplingFactors*", each of which contains 4 bits. They should be defined as:

```
<xs:element name="horizontalSamplingFactors"
   type="xs:byte">
   <xs:annotation>
      <xs:appinfo>
         <dfdl:bitstream length="4"/>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

```
<xs:element name="verticalSamplingFactors"
   type="xs:byte">
   <xs:annotation>
      <xs:appinfo>
         <dfdl:bitstream length="4"/>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

According to the definition, both of the two data segments are *xs:byte* data on the abstract level and contain 4 bits of the original physical byte. After that, when reading the actual image file where the original whole byte for them is 00100001 (0x21), their values can be interpreted as 00000010 (0x02) and 000000001 (0x01) respectively.

The bit streams in JPEG have some characteristics: all of them are 4 bits and there must be two consecutive 4-bit segments which make up a whole byte. It would be extremely complicated to process bit steams which cannot make up whole bytes, like 3 bits, 2 bits, and then 6 bits recursively, therefore my implementation has a restriction on bits segments: each consecutive pair of segments must compose a whole byte, like 4 bits + 4 bits or 3 bits + 5bits.

6.1.5 Variable length segments

Some data segments in JPEG have variable length, which is another challenge for the DFDL description. Theoretically, there are two kinds of variable length segment. One of them is a type of segment that has different length in different images, but the length always has a specific value which can be determined or calculated from the content of other segments; while the other kind of variable length segment does not have a tangible length, which is called "*unbounded*".

To define the first type of variable length segments, an XPath expression surrounded with a pair of curly brace "{" "}" will be specified as the value of the "*byteSize*" property of "*dfdl:dataFormat*" directive. The expression may contain any constant values and variables whose names are that of the existing elements it refers to. For example, the length of the quantisation table information segment "*QTInfo*" is equal to the value of "*lengthOfQT*" element minus 2, so the definition should be:

```
<xs:element name="lengthOfQT" type="xs:short"/>

<xs:element name="QTInfo" type="xs:byte">
   <xs:annotation>
      <xs:appinfo>
         <dfdl:dataFormat byteSize="{lengthOfQT-2}"/>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

When the "*byteSize*" is "unbounded", there are several topics that must be considered. Firstly, the unbounded element itself must only appear at most once, otherwise it is impossible to identify the end of the first element and the beginning of the second. Secondly, whether it appears and, if it appears, where the end of the segment is depends on the fixed value or candidate values enumeration of the next data element. If the next data segment does not have fixed or candidate values, the "*unbounded*" byte size definition becomes an ambiguous definition that is not allowed. However, the term "next data element" refers to a different element in a different situation, which is classified and listed in Figure 21

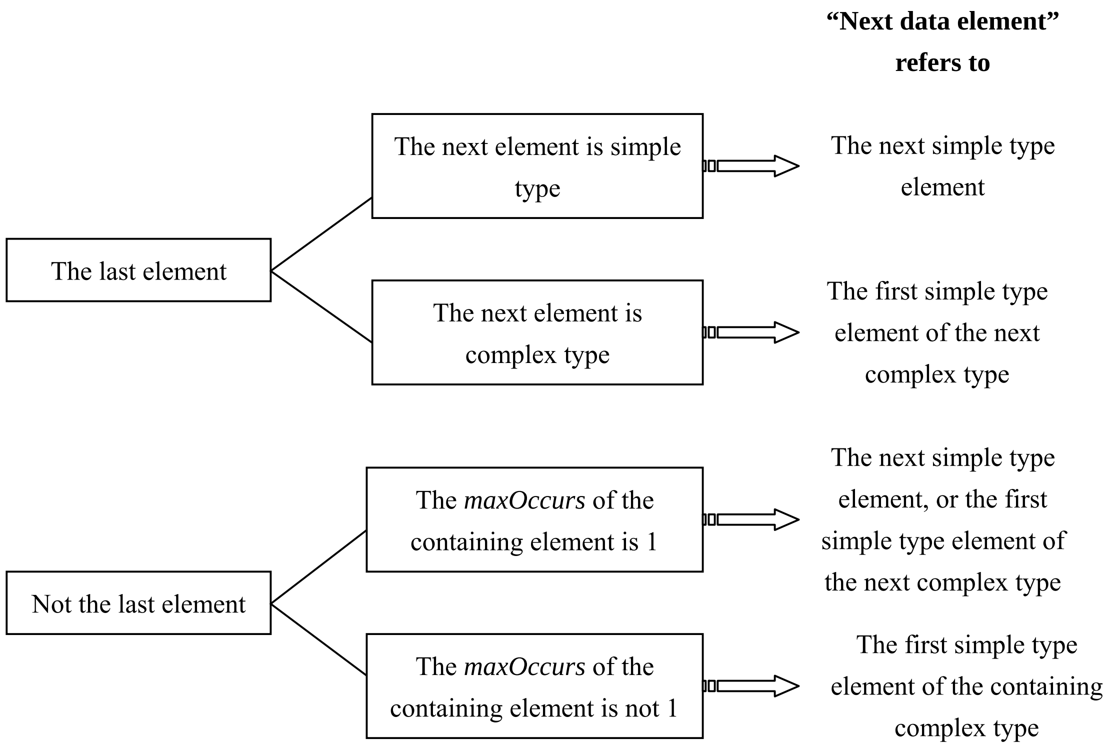

"Next data element" refers to
The last element
The next element is simple type
The next simple type element
The next element is complex type
The first simple type element of the next complex type
Not the last element
The maxOccurs of the containing element is 1
The next simple type element, or the first simple type element of the next complex type
The maxOccurs of the containing element is not 1
The first simple type element of the containing complex type

Figure 21 Possible situations of "next data element" referring to

To avoid confusion, I introduced a new property *terminator* of the "*dfdl:dataFormat*" directive, which is described in section 6.1.2.2. For example, as the entropy-coded segment is the last element of a *scanType* element and the scan may appear more than once, the candidate terminators must be explicitly specified:

```
<xs:element name="lastEntropyCodedSegment" type="xs:byte">
   <xs:annotation>
      <xs:appinfo>
         <dfdl:dataFormat byteSize="unbounded"/>
         <dfdl:dataFormat terminator="[0xffda, 0xffc0, 0xffd9]"/>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

If there is no confusion, in other words, there is only one possible terminator of the unbounded segment, the *terminator* annotation can be left out. For example, the *ignorableBytes* element of the *APP1Type* is also unbounded, but as the *maxOccurs* of the containing type *APP1Type* is 1, the next element must be the *frameType* in which the first simple element has fixed value 0xffdb, so it can be defined without specifying the terminator explicitly:

```
<xs:element name="ignorableBytes" type="xs:byte">
   <xs:annotation>
      <xs:appinfo>
         <dfdl:dataFormat byteSize="unbounded"/>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

6.2 The DFDL schema parser

6.2.1 Overall idea

According to the overall syntax, a DFDL Schema can be considered as a node tree composed of different purposed definition blocks, which should be processed separately. The typical tree representation of DFDL schema is shown in Figure 22

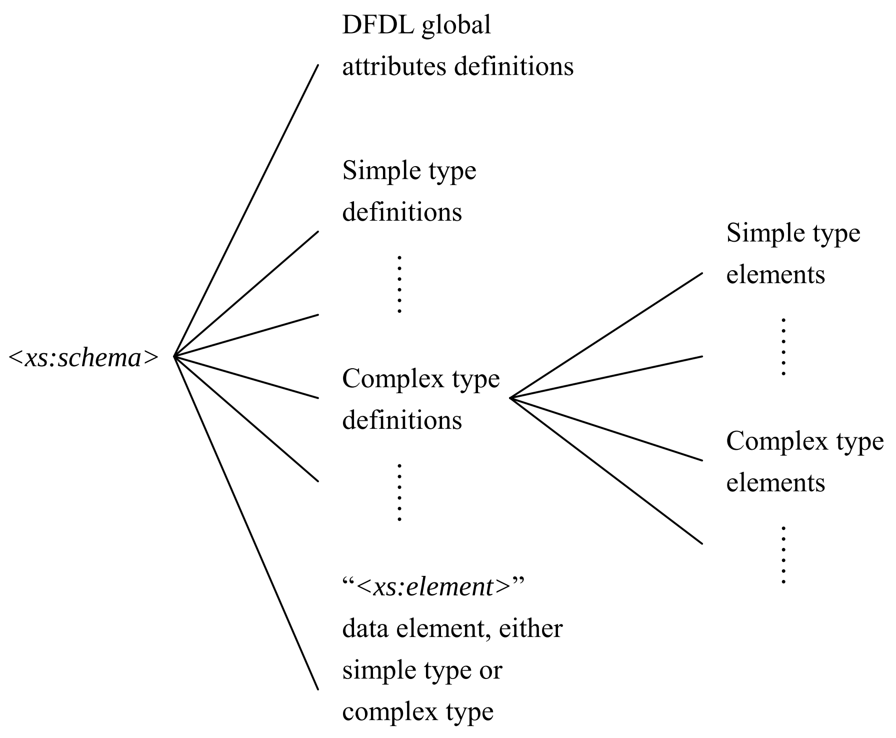


Figure 22 The typical tree representation of DFDL schema

The root of the tree is always "*<xs:schema>*" tag, indicating that this is an XML schema. Its first child is the DFDL global attributes definitions block, which is not compulsory; and then there are a number of simple type definitions and complex type definitions; finally it is a top-level "*<xs:element>*" data element, which is the root element of the target XML file. The order of the second level nodes is not fixed, in other words, the type definitions may appear after the element definition that refers to that type. On the lower level of the tree, complex type definition may contain an ordered sequence of simple type elements and complex type elements, and again each complex type element may contain the same kinds of children on and on. After classifying different kinds of definition blocks, they can be identified by using the DOM XML parser and then processed separately. Finally their structure information is stored into the appropriate part of the structure bean.

6.2.2 SAX vs. DOM

The entire parsing process is based on the Java DOM parser which is a standard way to parse XML files. However, at the beginning of the project, there was a choice between SAX and DOM, both of which are widely used XML parser APIs. The SAX APIs are very fast and suitable for parsing through a large XML file once, but are more difficult to use and its usability is limited. For example, it does not support the management of several XML elements at the same time or the ability to go back to the previous one etc. The DOM APIs allows the programmer to manage the entire document and provides them with more flexibility, but its scalability is limited by the length of the document and the memory of the computer, as it needs to store the entire document in a tree structure within the computer's memory. As a DFDL schema is usually not a very big file, but has a complicated structure that needs flexible processing, I chose to use the DOM APIs in my project.

6.2.3 Processing global DFDL attributes

After converting the DFDL Schema to a DOM tree, we traverse through its child nodes and process them one by one. The first child we might meet is the global DFDL attribute definitions block with "<*xs:annotation*>", "<*xs:appinfo*>" and "<*dfdl:definitions*>" tags.

Inside the <*dfdl:definitions*> block, there might be a number of attribute declarations. If it is an ordinary attribute, like representation type or byte order, the attribute name and its value will be stored into the attributes table, a *HashMap* object, of the structure bean. As we will need to frequently lookup attributes in the table later, the speed of access is significant for the program. A *HashMap* object is used because it provides very fast data lookup operations, although it sacrifices memory for the sake of speed.

A *HashMap* is conceptually a contiguous section of memory with a number of addressable elements [20] and it also can be thought of as a table with two columns. There are two sets of members in it, keys and values. When putting a key and a value into a *HashMap*, it will firstly performs a hashing function on the key object to map the key to a hash value, which is used as an index into the hash map identifying where to place the value in the memory. Similarly, when retrieving a value by specifying a particular key, the key will firstly be hashed to hash value, according to which the requested value can be located quickly.

For the same reason, both simple type definitions and complex type definitions are stored in *HashMap* objects. However, they are different in that the values of attributes table are strings specifying the attributes like "*binary*" or "*bigEndian*"; the values of simple types table are again *HashMap* objects containing the detailed properties of this type; while the values of complex types table are node trees, indicating what elements they consist of.

Besides ordinary attributes, there might be a couple of <*dfdl:use*> directives which place physical representation restrictions on the abstract data types. The first step is to check whether the mapped type has been registered in the simple types table. If it has been registered, simply link this mapped type with the abstract type, indicating that whenever we meet an element of this abstract type, it should be processed as a element of the mapped type. If it has not been registered, locate the mapped type definition within the whole DFDL description document, process and registered this simple type, and then link this mapped type with the abstract type.

What we mean by "linking" these two types is to insert a record to the global attributes table of the structure bean. However, it is different from inserting an ordinary attribute in that what we actually insert is the abstract type name, which is the key, and the mapped type name, which is the value.

6.2.4 Processing simple type definitions

After global attributes definitions, we need to process simple type definitions surrounded by "<*xs:simpleType*>" and "<*/xs:simpleType*>" tags. The challenge is that there might be two types of simple type definitions: the first one is independent definitions that exist on their own and have their own names; the other one is the definitions closely attached to particular elements. The program thus must be able to identify both of these two types of simple type definitions. What's more, the second type of definitions does not have to have a name attribute. I use the name of the element it attaches to as its name, because firstly this name can exactly denote what the type is, and secondly name confliction is avoided, in other words, the situation that two types have the same name but different type properties will not occur, as the elements with the same name always are same type.

Another interesting point of the simple type definitions is that the values of the simple type table are also *HashMap* objects. In the inner table, the keys are the names of different type properties and the values are the corresponding values. For example, the structure information of simple type "*binaryByte*" in the structure bean is shown in Figure 23

Key	Value	
dfdl:binaryByte	**Key**	**Value**
	base	xs:byte
	dfdl:byteSize	1
	dfdl:repType	binary

Figure 23 The structure information of simple type "*binaryByte*"

Finally, we should pay more attention to the "base" type. When the new type is based on an existing simple type, the properties of the existed type should be copied to the new type, and moreover, if the type is an abstract type, we should locate the proper mapped type and copy its properties. For example, the simple type element "*numberOfComponents*" has its own type definition which is based on "*xs:byte*", so we have to locate the existing "*dfdl:binaryByte*" type which is the mapped type of "*xs:byte*", copy its properties and then add new properties. The structure information of "*numberOfComponents*" type is shown in Figure 24

Key	Value	
numberOfComponents	**Key**	**Value**
	base	xs:byte
	dfdl:byteSize	1
	dfdl:repType	binary
	enumeration	[0x01, 0x03, 0x04]

Figure 24 The structure information of "*numberOfComponents*" type

6.2.5 Processing complex type definitions

Whenever the DOM parser arrives at the "*<xs:complexType>*" tag, the processing of a complex type should begin. At first, it has the same name problem with processing simple types and the same solution.

However, the most significant feature of processing complex types is that the structure information of a complex type is assembled into a node tree, which is then stored entirely into the complex types table of the structure bean, and each node is a structure item. As a complex type may consist of two types of elements, simple type and complex type, there should be two types of structure item corresponding to them, simple structure item and complex structure item. In addition, another type of structure item, which is called the label structure item, is required for constructing the tree. All structure items consist of *name*, *minOccurs*, *maxOccurs* and *fixed value* attributes of the corresponding data elements. Label structure items only contain these four attributes; while simple structure items and complex structure items contain another "*type*" attribute specifying what type the data element is.

The first step of processing the complex type is to construct the root node of the tree with a label structure item containing the type name, then look into the "*<xs:sequence>*" definition block and process each element. The result of processing an element will be a tree with only the root node containing the proper structure item, and then this result tree should be inserted into the complex type tree as a child of its root node. Taking the "*JpegDFDL*" complex type for example, its structure information is represented in Figure 25

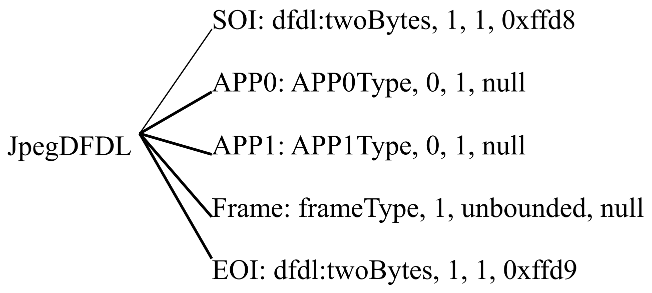
Key	Value
JpegDFDL	 JpegDFDL SOI: dfdl:twoBytes, 1, 1, 0xffd8 APP0: APP0Type, 0, 1, null APP1: APP1Type, 0, 1, null Frame: frameType, 1, unbounded, null EOI: dfdl:twoBytes, 1, 1, 0xffd9

Figure 25 The structure information of "*JpegDFDL*" complex type

6.2.6 Process elements

When processing elements, there are several different situations that should be focused on, resulting in a tree with only the root node containing the proper structure item being returned.

Firstly, if the element explicitly specifies a type through the "*type*" attribute, either a simple type or a complex type, and this type has been registered, a tree with a structure item of this type can be simply created and returned.

Secondly, if the "*type*" attribute of the element is explicitly specified but the type has not been registered, or if there is no explicit "*type*" attribute declaration but the type definition is immediately attached to the element, what we should do is to locate the type definition, process and register it appropriately at first, and then process the element as same as the first situation.

Finally, and also the most challenging situation, if there is an "*<xs:annotation>*" definition block immediately after the element to be processed, it means that there are some restrictions placed on the element's physical representation, and these restrictions should be added to the original properties of the base type, overriding any existing properties, such that a new simple type definition is formed and then registered. I decided to use

```
“the base type name” + “&” + “the element name”
```

as the name of the new type. However, the base type also may or may not have been registered. Thus we should correctly process and register the base type first if it does not exist, and then define the new type according to the DFDL annotation.

For example, the following code defines the “*bytesToBeRead*” data element,

```
<xs:element name="bytesToBeRead" type="xs:byte">
   <xs:annotation>
      <xs:appinfo>
         <dfdl:dataFormat
            byteSize="{thumbnailWidth*thumbnailHeight*3}"/>
      </xs:appinfo>
   </xs:annotation>
</xs:element>
```

placing a new restriction on the byte size attribute of the “*xs:byte*” type via DFDL annotation. Consequently, the structure information of this new type should be:

Key	**Value**	
	Key	**Value**
dfdl:binaryByte&bytesToBeRead	base	xs:byte
	dfdl:byteSize	{thumbnailWidth*thumbnailHeight*3}
	dfdl:repType	binary

Figure 26 The structure information of a new type defined by DFDL annotation

After defining the new type, the element can be processed as same as the first situation.

6.3 The data file parser

6.3.1 Recursion algorithm

After the structure bean is generated by the DFDL Schema parser, all structure information required for parsing the data file and extracting individual data elements has been well prepared. The core algorithm of parsing the data file is recursive, and its pseudo-code is shown in Figure 27

```
Traverse the structure tree {
    IF  (it is a simple structure item) {
        Process simple data element
    }
    ELSE IF (it is a complex structure item) {
        Process complex data element
    }
}
```

Process simple data element:

```
Prepare necessary parameters
Generate the local data tree

FOR (i = 1 to minOccurs) {
    Read the data;
    Add the data to the data tree;
}

WHILE (i < maxOccurs) AND (it is not the end of the
    data file) {
    IF (this data element does not occur) BREAK
    Read the data
    Add the data to the data tree
    i++
}

RETURN the result data tree
```

Process complex data element:

```
Prepare necessary parameters
Generate the local data tree

FOR (i = 1 to minOccurs) {
    Traverse the complex type tree {
        IF  (it is a simple structure item) {
            Process simple data element
        }
        ELSE IF (it is a complex structure item) {
            Process complex data element
        }
        Merge the result tree to the data tree
    }
}

WHILE (i < maxOccurs) AND (it is not the end of the
    data file) {
    IF (this data element does not occur) BREAK;
    Traverse the complex type tree {
        IF  (it is a simple structure item) {
            Process simple data element
        }
        ELSE IF (it is a complex structure item) {
            Process complex data element
        }
        Merge the result tree to the data tree
    }
    i++
}

RETURN the result data tree
```

Figure 27 The pseudo-code of the recursion algorithm for parsing data file

On the highest level, the structure tree of the structure bean is traversed and its children are properly processed according to their types. During processing the data element, both simple type and complex type, the difficulty is that one structure item may correspond to several actual data elements, which is determined by the *minOccurs* and *maxOccurs* attribute of the structure item. The *minOccurs* attribute indicates the number of times this data element must appear, so the *FOR* structure is used to iteratively process the data element *minOccurs* times; however, as there is no indication how many times the data element actually occurs, we have to perform the same processing repeatedly, checking every time that it really appears, as long as the number of times we have tried is still less than the *maxOccurs*.

The result of processing a simple element is a tree whose children nodes are *SimpleDataItem*s containing the element name, value, and some attributes. These children nodes will be later merged to the parent data tree. If a data element appears 4 times, for example, the result tree looks like:

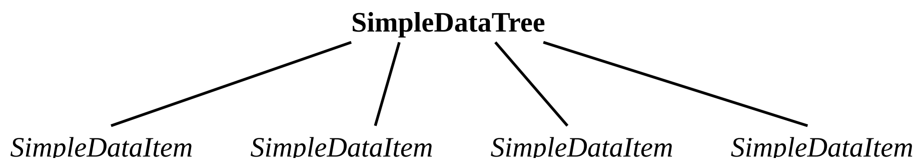


Figure 28 Simple data tree

Finally, processing complex data elements is the core of the recursion: the complex type tree is traversed; if the node is a simple structure item, process the simple element as above; if it is a complex structure item, process it as another complex data element; and merge all of these result trees to a parent complex data tree. Take the complex date element "*frameHeader*", whose structure is shown in Figure 29a, as an example,

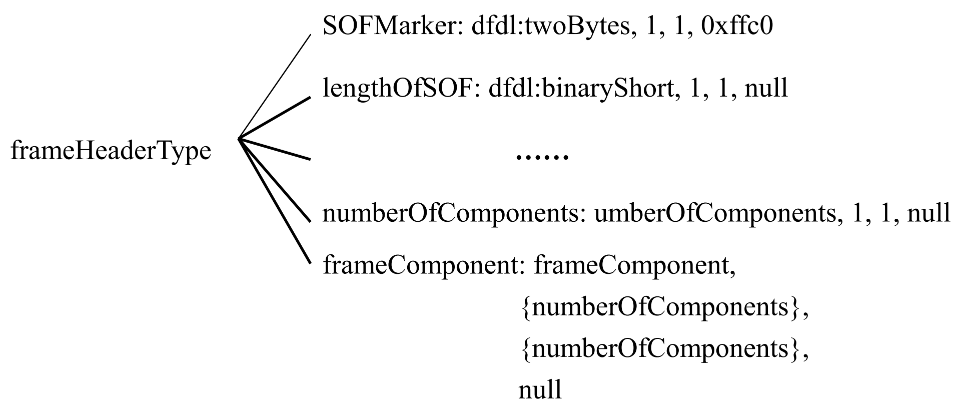

frameHeaderType
SOFMarker: dfdl:twoBytes, 1, 1, 0xffc0
lengthOfSOF: dfdl:binaryShort, 1, 1, null
……
numberOfComponents: umberOfComponents, 1, 1, null
frameComponent: frameComponent, {numberOfComponents}, {numberOfComponents}, null

Figure 29a The structure of "*frameHeader*" element

it consists of several simple type elements and a complex type element "*frameComponent*" which comprises another four simple type elements and may appear more than once. If it appears two times for instance, the complex data tree for "*frameComponent*" elements will consist of two branches, each of which is a "*frameComponent*", see Figure 29b.

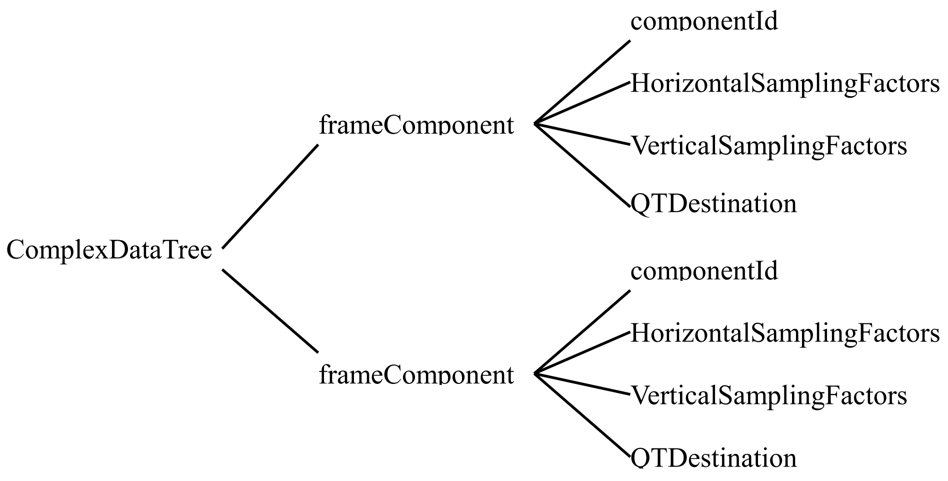


Figure 29b The complex data tree for "*frameComponent*" data elements

Finally, this complex data tree as well as other simple data trees will be merged to the parent "*frameHeader*" complex data tree shown in Figure 29c

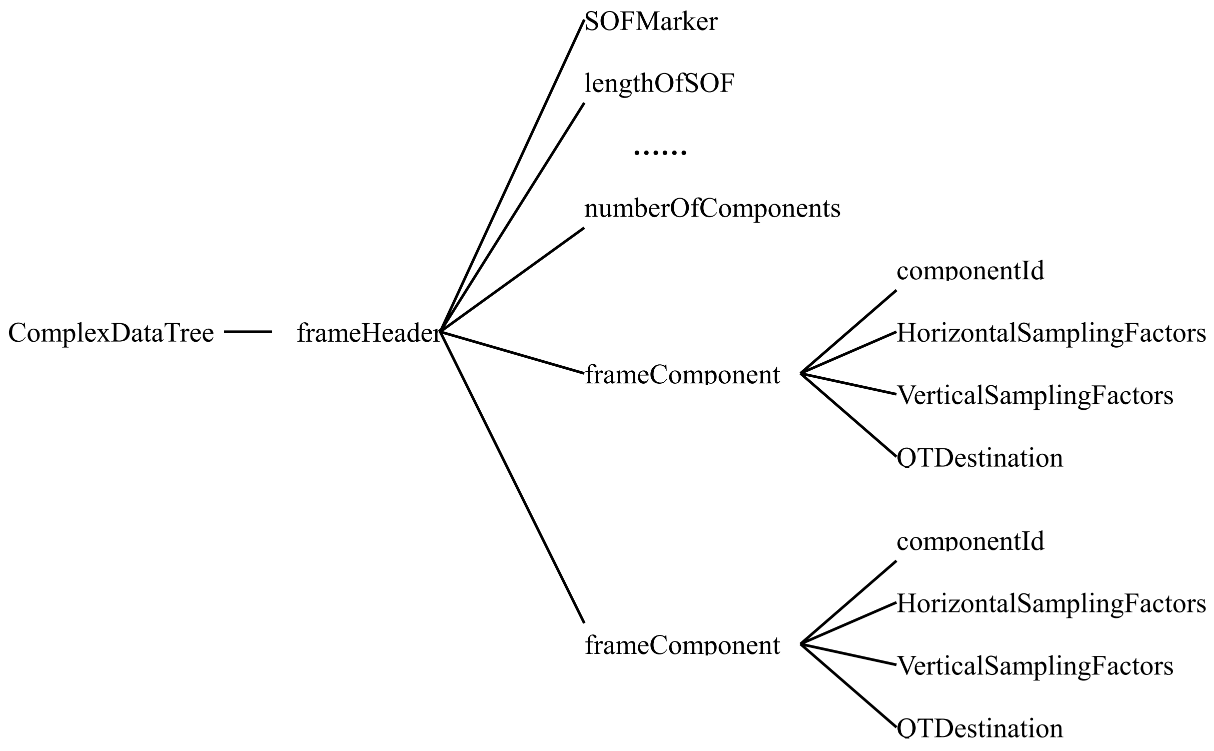


Figure 29c The complex data tree for “*frameHeader*” data elements

6.3.2 File wrapper

Whilst parsing the data file, we must frequently read the data file and as the reading process is sometimes quite complicated, I designed a *FileWrapper* class encapsulating a number of powerful reading methods, which can be used easily and flexibly during parsing the data file.

For instance, one of the read methods, *readUntil()*, is critical for parsing variable length data segments which are unbounded. As data element is unbounded, we have to determine how many bytes to read before reading it. Either a terminator or a collection of possible terminators can help to determine the length of the data element, so there are two *readUntil()* methods with different parameters based on the Java polymorphism and override mechanism. Both of them firstly lookup the appropriate terminator in the data file, calculate how many bytes should be read and then read the data.

6.3.3 Convertor

Convertor is another very important and useful utility class designed for facilitating the parsing of data file. Generally speaking, there are two categories of methods: one is for parsing the XPath expression, and the other one is for the conversion between bytes array and different representations like integer, string etc.

As mentioned before, one of the challenges of this project is to process variable length data segments, including unbounded segments and segments whose lengths depend on the contents of other segments. By using the *readUntil()* method of file wrapper, unbounded segments can be parsed. However, for parsing the other type of variable length segments, it is required to be able to evaluate their byte size attribute that is an XPath expression. The *expToValue()* method of *Convertor* class solve this problem.

The first step is to format the expression string as there might be "{", "}" and spaces which are illegal characters in an expression. Secondly, we must separate variables and constants from the expression. The values of variables come from the contents of other data segments, so I designed a lookup table storing the names and values of all data elements that have been parsed such that values of those required variables could be found from this lookup table. Finally, the Java Mathematical Expression Parser [21] library is used to evaluating mathematical expressions of the "*byteSize*" attribute.

In addition, there is another series of methods which operate on the bytes array, for instance, converting the bytes array to the corresponding integer value based on a particular byte order, or representing the array as a string of hex numbers etc. These methods are also useful when generating the XML representation from the data bean.

6.4 Testing

6.4.1 Unit testing

The whole system, especially the two parsers, is quite complex, requiring several levels of method invoking. Therefore, it is of great importance to ensure that the lowest level methods work correctly before implementing the method that will invoke them. Otherwise, it would be extremely difficult to locate errors. Moreover, the developer is always the person who understands the program best, so it is much more straightforward to test the program during development. Unit testing introduces the concept of writing independent units in isolation from the whole system to test the correctness of particular modules of the code. It helps the programmers to become more productive, while at the same time increasing the quality of the developed code [22]. When unit testing is utilized, it should be part of the development rather than after the development. Nowadays, the notion of "test first" has become more and more popular: writing unit tests before writing the code to be tested. By doing this, the test puts the entire focus on whether the result is what is expected, and it will not be affected by the actual implementation.

Junit is a freely available framework for unit testing, providing a wide range of useful methods which minimised the required work. It is used in my project for testing some critical classes, like the parsers and different utilities classes. Taking the *bytesToText()* method from the *Convertor* class as an example, it aims to convert a byte array into a text string based on the ASCII character codes. If the byte array is

```
0x45, 0x78, 0x69, 0x66
```

the expected result should be "Exif". So in the test case, both the original array and the expected result are specified, and then the *assertEquals()* provided by Junit is invoked to test whether the result of the function is equal to the expected one. After coding the test, I implemented the *bytesToText()* method and then tested this particular module separately until it works correctly.

Within the system, there are some classes that do not need to be tested. For instance, the interface and abstract classes, which do not have actual implementation, and the data model classes which only have simple *get* and *set* methods. Amongst all the classes, I chose to test the utility classes including *Convertor*, *FileWrapper*, *SchemaUtilities* and *DataParserUtilities*; the builder classes like *StructureBeanBuilder*, *DataBeanBuilder* and *XMLBuilder*; and the higher-level classes including *SimpleDFDLParser*, *SimpleDataFileParser* and *DFDLLibrary*. I chose them because they are the actual implementation of the core functions of the system.

6.4.2 System testing

Unit testing is used for testing individual modules of the system. However, whether these modules can work together and produce the right result still needs to be tested. So besides Junit test cases, I designed three system-test classes to test if the whole system works well with a binary file, a text file and a JPEG image.

- The binary file contains three groups of binary data, and each group comprises of a byte, a short, an integer, a long, a float and a double. The binary sequence is:

  ```
  01 00 01 00 00 00 01 00 00 00 00 00 00 00 01 3F 80 00
  00 3F F0 00 00 00 00 00 00 02 00 02 00 00 00 02 00 00
  00 00 00 00 00 02 40 00 00 00 40 00 00 00 00 00 00 00
  03 00 03 00 00 00 03 00 00 00 00 00 00 00 03 40 40 00
  00 40 08 00 00 00 00 00 00
  ```

  The DFDL description for this binary data file and the result XML representation are shown in Appendix Ⅰ.

- The text file contains five students' records, each of which comprises of "ID" (10 characters), "Name" (20 characters) and "Sex" (6 characters) three fields. It looks like:

```
0506004   Yi Zhu            Male
0508002   Merry             Female
0602010   James             Male
0455504   Haogang Zhu       Male
0309111   Yaoyao Liu        Female
```

  The DFDL description for this text file and its result XML representation are shown in Appendix Ⅱ.

- The structure of the JPEG image has been described above, so I will not repeat here. The DFDL Schema for JPEG data format and the result XML representation of one example image is given in Appendix Ⅲ.

Within each system-test class, I called the two parsers one after the other, taking the DFDL description and the data file as their inputs respectively, and afterward generated the XML representation of the file and checked the result by human comparison.

There are some other methods that could be used to check the resulting XML representations. The first one is that we could generate an expected XML representation by hand, which can then be used for automatic comparison with the resulting one. However, the contents of the JPEG images are too complex to do so. Alternatively, we could design another parser for parsing the XML representation and reverting it to the binary representation, and then compare the resulting binary file with the original one. This method is powerful, credible and very easy to use, but developing the new parser will be another complicated project and take a lot of time. Considering all the pros and cons, I decided to use human comparison at this stage, which is the most straightforward method for this project.

Finally, I run the same system tests on three platforms: Windows XP, Sun Solaris 8 and Redhat Linux 9, and they produced the correct results.

6.5 Summary

In the implementation phase, I successfully described the JPEG data format as well as another two types of data file based on a subset of the DFDL specification. I designed a generic DFDL parser for parsing any DFDL descriptions based on that subset of the DFDL specification, and a data file parser for parsing the corresponding data file, and finally an XML representation of the original data file will be produced. All of these functionalities work correctly with three types of data file on three different platforms.

However, whilst trying to describe the JPEG format, I found some limitations to the DFDL. Firstly, it cannot express condition dependence. JPEG is not a context free data format. Sometimes, the length of a data segment or the number of times it actually appears directly depends on the contents of other segments, and this situation has been handled. However, sometimes whether the data segment appears depends on the value of a particular segment. For instance, if the restart interval value of DRI (Define restart interval) segment is zero or this marker segment does not appear at all, it means that restart intervals for the following scans are disabled and there will be only one entropy-coded segment; otherwise if the value is nonzero, the restart intervals are enables and there might be several entropy-coded segments. DFDL is not powerful enough to define this condition relationship. Secondly, DFDL cannot deal with indirect reference. In the APP1 definition segment of JPEG, sometimes the value of an entry is not the actual data but the relative address of the data, and moreover whether or not it is the data depends on the value itself. This kind of reference is too complex for DFDL to define.

Chapter 7 Conclusion

In this project, I utilised a subset of the DFDL specification to successfully solve a complicated real world problem – describing the JPEG data format, and then implemented a DFDL library, including a generic DFDL Schema parser for parsing any DFDL descriptions and a data file parser for extracting individual data segments according to the structure information provided by the DFDL Schema parser, and finally the original JPEG data file can be represented in XML format. All of these have met our original goals – determining how to describe the JPEG data format using a subset of the DFDL specification, and implementing a generic parser and some other libraries, which result in an XML representation of the original data file.

One significant characteristic of this project is that it follows formal software engineering principles and uses a series of development processes, which highly increase the chance of project success. The most significant and valuable one is the risk management. As I was not familiar with DFDL at the beginning of the project, I seriously considered it as a potential risk, which would highly affect the entire project, and thought of some possible solutions. This risk became real a few weeks into the project. At first, I thought that I should write DFDL descriptions for individual JPEG images, however, after the discussion with DFDL working group, I realised that I should define a DFDL description for the JPEG data format rather than individual images. Because of this timely correction, the misunderstanding did not significantly affect the project. In addition, my choice of “Design to schedule” development model and my work plan were shown to be reasonable; I followed them quite well and finished all deliverables on time.

Furthermore, I learned and tried to use a variety of techniques which are absolutely new for me in the project, like the unit test and different design patterns. I wrote unit tests for some core classes, for instance the *Convertor* and the *FileWrapper* etc., before coding. It made sure that those utility methods would work correctly, which provided a solid foundation for the implementation of the parsers. Design patterns like “builder” help to simplify the entire system design and increase the robustness of the code.

The DFDL Schema for JPEG data format I designed would be a valuable example of using DFDL to solve practical problems, and would be very helpful to other DFDL learner in the future, as when I studied DFDL, the most significant difficulty was that there was no example of describing practical problems using DFDL. Furthermore, my implementation of the DFDL Schema parser and the data file parser introduces a notion of using tree data structure to keep the structure information defined in the DFDL Schema and the actual data from the original data file. Last but not least, whilst coming up with the DFDL description for the JPEG data format, I found some limitation of the DFDL specification, including expressing condition dependence and indirect reference, which would also be very valuable for the future DFDL research.

In a nutshell, the entire project completed successfully and I learned a lot of new things from it. I am sure it would be helpful to the entire DFDL project, because I not only utilised the DFDL to solve a real problem, but also tried to overcome some challenges, and what’s more, I found a few limitations to the DFDL. As to these limitations, further investigation will be required in the future, and I hope to be able to carry out some of this work.

Reference

[1] Grid.org, Grid Computing: The Evolution, http://www.grid.org/about/gc/evolution.htm

[2] Ian Foster, Carl Kesselman, The Grid 2: Blueprint for a New Computing Infrastructure

[3] Mario Antonioletti, Malcolm Atkinson, Rob Baxter, Andrew Borley, Neil Chue Hong, Brian Collins, Neil Hardman, Alastair C. Hume, Alan Knox, Mike Jackson, Amy Krause, Simon Laws, James Magowan, Norman W. Paton, Dave Pearson, Tom Sugden, Paul Watson, Martin Westhead, The design and implementation of Grid database services in OGSA-DAI, Concurrency and Computation: Practice and Experience Volume 17, Issue 2-4 , Pages 357 - 376, Feb 2005.

[4] DFDL working group webpage, http://forge.gridforum.org/projects/dfdl-wg

[5] Mike Beckerle, Martin Westhead, GGF DFDL Primer, https://forge.gridforum.org/projects/dfdl-wg/document/DFDL_Primer/en/1

[6] W. Fenner, R. Frederick, RTP Payload Format for JPEG-compressed Video, http://xml.resource.org/public/rfc/html/rfc2035.html

[7] JPEG File Interchange Format Version 1.02, http://www.jpeg.org/public/jfif.pdf

[8] International standard DIS 10918-1, CCITT recommendation T.81, Digital Compression and Coding of Continuous-tone Still Images

[9] Introduction to HDF5, http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.intro.html

[10] BinX, http://www.edikt.org/binx/

[11] William B. Pennebaker, Joan L. Mitchell, JPEG – still image data compression standard, Chapter 2, P9

[12] William B. Pennebaker, Joan L. Mitchell, JPEG – still image data

compression standard, Chapter 2, P13

[13] William B. Pennebaker, Joan L. Mitchell, JPEG – still image data compression standard, Chapter 7, P97

[14] JPEG File Layout and Format, http://www.funducode.com/freec/Fileformats/format3/format3b.htm

[15] Cay S. Horstmann, Gary Cornell, Core Java 2: Volume Ⅱ - Advanced Fetures, Chapter 8

[16] http://www.jdsl.org/

[17] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: elements of reusable object-oriented software, Chapter 3, P97

[18] Kristoffer H. Rose, DFDL Proposal

[19] W3C Recommendation, XML Schema Part 0: Primer, http://www.w3.org/TR/xmlschema-0/

[20] Ben Tindale, Hash table in Java, http://www.linuxgazette.com/issue57/tindale.html

[21] http://www.singularsys.com/jep/

[22] Keld H. Hansen, Unit Testing Java Programs, http://javaboutique.internet.com/tutorials/UnitTesting/

Appendix Ⅰ Describing a simple binary data file

➢ The DFDL description (SimpleBinary.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace=http://dataformat.org/
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:dfdl="http://dataformat.org/">

  <xs:annotation>
    <xs:appinfo>
      <dfdl:definitions>
        <dfdl:dataFormat repType="binary"/>
        <dfdl:dataFormat byteOrder="littleEndian"/>
        <dfdl:use type="dfdl:binaryByte"/>
        <dfdl:use type="dfdl:binaryShort"/>
        <dfdl:use type="dfdl:binaryInt"/>
        <dfdl:use type="dfdl:binaryLong"/>
        <dfdl:use type="dfdl:binaryFloat"/>
        <dfdl:use type="dfdl:binaryDouble"/>
      </dfdl:definitions>
    </xs:appinfo>
  </xs:annotation>

  <xs:simpleType name="dfdl:binaryByte" dfdl:byteSize="1"
    dfdl:repType="binary">
    <xs:restriction base="xs:byte"/>
  </xs:simpleType>

  <xs:simpleType name="dfdl:binaryShort" dfdl:byteSize="2"
    dfdl:repType="binary">
    <xs:restriction base="xs:short"/>
  </xs:simpleType>

  <xs:simpleType name="dfdl:binaryInt" dfdl:byteSize="4"
    dfdl:repType="binary">
    <xs:restriction base="xs:int"/>
  </xs:simpleType>

  <xs:simpleType name="dfdl:binaryLong" dfdl:byteSize="8"
    dfdl:repType="binary">
    <xs:restriction base="xs:long"/>
  </xs:simpleType>

  <xs:simpleType name="dfdl:binaryFloat" dfdl:byteSize="4"
    dfdl:repType="binary">
    <xs:restriction base="xs:float"/>
  </xs:simpleType>

  <xs:simpleType name="dfdl:binaryDouble" dfdl:byteSize="8"
    dfdl:repType="binary">
    <xs:restriction base="xs:double"/>
  </xs:simpleType>

  <xs:element name="SimpleBinary">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="DataSet" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="byte" type="xs:byte"/>
              <xs:element name="short" type="xs:short"/>
              <xs:element name="int" type="xs:int"/>
              <xs:element name="long" type="xs:long"/>
              <xs:element name="float" type="xs:float"/>
              <xs:element name="double" type="xs:double"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

- The XML representation of the original data file (SimpleBinaryXML.xml)

```
<DFDL>
  <SimpleBinary>
    <DataSet>
      <byte>01</byte>
      <short>0001</short>
      <int>00000001</int>
      <long>0000000000000001</long>
      <float>3f800000</float>
      <double>3ff0000000000000</double>
    </DataSet>
    <DataSet>
      <byte>02</byte>
      <short>0002</short>
      <int>00000002</int>
      <long>0000000000000002</long>
      <float>40000000</float>
      <double>4000000000000000</double>
    </DataSet>
    <DataSet>
      <byte>03</byte>
      <short>0003</short>
      <int>00000003</int>
      <long>0000000000000003</long>
      <float>40400000</float>
      <double>4008000000000000</double>
    </DataSet>
  </SimpleBinary>
</DFDL>
```

Appendix Ⅱ Describing a simple text file

➢ The DFDL description (SimpleText.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace=http://dataformat.org/
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:dfdl="http://dataformat.org/">

  <xs:annotation>
    <xs:appinfo>
      <dfdl:definitions>
        <dfdl:dataFormat repType="text"/>
        <dfdl:dataFormat characterSet="UTF-8"/>
        <dfdl:use type="dfdl:binaryByte"/>
        <dfdl:use type="dfdl:textString"/>
      </dfdl:definitions>
    </xs:appinfo>
  </xs:annotation>

  <xs:simpleType name="dfdl:binaryByte" dfdl:byteSize="1"
    dfdl:repType="binary">
    <xs:restriction base="xs:byte"/>
  </xs:simpleType>

  <xs:simpleType name="dfdl:textString" dfdl:byteSize="unbounded">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>

  <xs:element name="SimpleText">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Student" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>

              <xs:element name="ID" type="xs:string">
```

```
                <xs:annotation>
                  <xs:appinfo>
                    <dfdl:dataFormat byteSize="10"/>
                  </xs:appinfo>
                </xs:annotation>
              </xs:element>

              <xs:element name="Name" type="xs:string">
                <xs:annotation>
                  <xs:appinfo>
                    <dfdl:dataFormat byteSize="20"/>
                  </xs:appinfo>
                </xs:annotation>
              </xs:element>

              <xs:element name="Sex" type="xs:string">
                <xs:annotation>
                  <xs:appinfo>
                    <dfdl:dataFormat byteSize="6"/>
                  </xs:appinfo>
                </xs:annotation>
              </xs:element>

              <xs:element name="CR" type="xs:byte" fixed="0x0d"/>

              <xs:element name="LF" type="xs:byte" fixed="0x0a"/>

            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

➢ The XML representation of the original text file (SimpleTextXML.xml)

```
<DFDL>
  <SimpleText>
    <Student>
      <ID>0506004   </ID>
      <Name>Yi Zhu            </Name>
      <Sex>Male  </Sex>
      <CR>0d</CR>
      <LF>0a</LF>
    </Student>
    <Student>
      <ID>0508002   </ID>
      <Name>Merry             </Name>
      <Sex>Female</Sex>
      <CR>0d</CR>
      <LF>0a</LF>
    </Student>
    <Student>
      <ID>0602010   </ID>
      <Name>James             </Name>
      <Sex>Male  </Sex>
      <CR>0d</CR>
      <LF>0a</LF>
    </Student>
    <Student>
      <ID>0455504   </ID>
      <Name>Haogang Zhu        </Name>
      <Sex>Male  </Sex>
      <CR>0d</CR>
      <LF>0a</LF>
    </Student>
    <Student>
      <ID>0309111   </ID>
      <Name>Yaoyao Liu         </Name>
      <Sex>Female</Sex>
      <CR>0d</CR>
      <LF>0a</LF>
    </Student>
  </SimpleText>
</DFDL>
```

Appendix Ⅲ Describing a JPEG image

➢ The DFDL Schema for the JEPG format (JPEG_DFDL_Schema.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  targetNamespace="http://dataformat.org/"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:dfdl="http://dataformat.org/">

  <!--DFDL setup-->
  <xs:annotation>
    <!-- Configuration & defaults -->
    <xs:appinfo>
      <dfdl:definitions>
        <dfdl:dataFormat repType="binary"/>
        <dfdl:dataFormat byteOrder="bigEndian"/>
        <dfdl:use type="dfdl:twoBytes"/>
        <dfdl:use type="dfdl:binaryByte"/>
        <dfdl:use type="dfdl:binaryShort"/>
        <dfdl:use type="dfdl:binaryInt"/>
        <dfdl:use type="dfdl:textString"/>
      </dfdl:definitions>
    </xs:appinfo>
  </xs:annotation>


  <!-- Mapping for Binary types-->
  <xs:simpleType name="dfdl:twoBytes" dfdl:byteSize="2"
    dfdl:repType="binary">
    <xs:restriction base="xs:hexBinary"/>
  </xs:simpleType>

  <xs:simpleType name="dfdl:binaryByte" dfdl:byteSize="1"
    dfdl:repType="binary">
    <xs:restriction base="xs:byte"/>
  </xs:simpleType>
```

```
<xs:simpleType name="dfdl:binaryShort" dfdl:byteSize="2"
  dfdl:repType="binary">
  <xs:restriction base="xs:short"/>
</xs:simpleType>

<xs:simpleType name="dfdl:binaryInt" dfdl:byteSize="4"
  dfdl:repType="binary">
  <xs:restriction base="xs:int"/>
</xs:simpleType>

<xs:simpleType name="dfdl:textString" dfdl:repType="text">
  <xs:restriction base="xs:string"/>
</xs:simpleType>

<!-- DFDL description of the JPEG format-->
<xs:element name="JpegDFDL">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="SOI" type="xs:hexBinary" fixed="0xffd8"/>
      <xs:element name="APP0" type="APP0Type" minOccurs="0"/>
      <xs:element name="APP1" type="APP1Type" minOccurs="0"/>
      <xs:element name="frame" type="frameType"
          maxOccurs="unbounded"/>
      <xs:element name="EOI" type="xs:hexBinary" fixed="0xffd9"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>


<!--Definitions of different segments-->

<xs:complexType name="APP0Type">
  <xs:sequence>
    <!-- JFIF header -->
    <xs:element name="APP0Marker" type="xs:hexBinary"
       fixed="0xffe0"/>
    <xs:element name="lengthOfAPP0" type="xs:short"/>
    <xs:element name="fileId1" type="xs:string" fixed="JFIF"/>
    <xs:element name="fileId2" type="xs:byte" fixed="0x00"/>

    <!-- JFIF data -->
    <xs:element name="MajorRevisionNumber" type="xs:byte"/>
    <xs:element name="MajorRevisionNumber" type="xs:byte"/>
    <xs:element name="unitsForDensities">
       <!-- 0 = no units, x/y density specify the aspect ratio instead
       -->
       <!-- 1 = x/y density are dots/inch -->
       <!-- 2 = x/y density are dots/cm -->
       <xs:simpleType>
         <xs:restriction base="xs:byte">
           <xs:enumeration value="0x00"/>
           <xs:enumeration value="0x01"/>
           <xs:enumeration value="0x02"/>
         </xs:restriction>
       </xs:simpleType>
    </xs:element>
    <xs:element name="xDensity" type="xs:short"/>
    <xs:element name="yDensity" type="xs:short"/>
    <xs:element name="thumbnailWidth" type="xs:byte"/>
    <xs:element name="thumbnailHeight" type="xs:byte"/>

    <!-- n bytes for thumbnail, n = width*height*3 bytes -->
    <xs:element name="bytesToBeRead" type="xs:byte">
      <xs:annotation>
       <xs:appinfo>
         <dfdl:dataFormat byteSize="{thumbnailWidth*thumbnailHeight
           *3}"/>
       </xs:appinfo>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="APP1Type">
  <xs:sequence>
    <!-- Exif header -->
    <xs:element name="APP1Marker" type="xs:hexBinary"
       fixed="0xffe1"/>
    <xs:element name="lengthOfAPP1" type="xs:short">
      <xs:annotation>
        <xs:appinfo>
          <dfdl:dataFormat byteOrder="littleEndian"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
```

```
<xs:element name="fileId1" type="xs:string" fixed="Exif">
  <xs:annotation>
    <xs:appinfo>
      <dfdl:dataFormat byteSize="4"/>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
<xs:element name="fileId2" type="xs:byte" fixed="0x00"/>
<xs:element name="fileId3" type="xs:byte" fixed="0x00"/>

<!-- TIFF header -->
<xs:element name="byteAlign" type="xs:hexBinary"/>
<xs:element name="tagMark" type="xs:short"/>
<xs:element name="offsetToIFD0" type="xs:int"/>

<!-- Image File directory of main image -->
<xs:element name="IFD0">
  <xs:complexType>
    <xs:sequence>
       <xs:element name="numberOfEntry" type="xs:short">
        <xs:annotation>
         <xs:appinfo>
           <dfdl:dataFormat byteOrder="littleEndian"/>
         </xs:appinfo>
       </xs:annotation>
      </xs:element>
      <xs:element name="entry" minOccurs="{numberOfEntry}"
        maxOccurs="{numberOfEntry}">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="tagNumber" type="xs:short"/>
            <xs:element name="dataFormat" type="xs:short"/>
            <xs:element name="numberOfEntryComponents"
                type="xs:int"/>
            <xs:element name="dataValue" type="xs:int"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
       <xs:element name="offsetToIFD1" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
    <!-- unstructured bytes containing values for tags -->
    <xs:element name="ignorableBytes" type="xs:byte">
      <xs:annotation>
          <xs:appinfo>
            <dfdl:dataFormat byteSize="unbounded"/>
          </xs:appinfo>
      </xs:annotation>
    </xs:element>

  </xs:sequence>
</xs:complexType>



<xs:complexType name="frameType">
  <xs:sequence>
    <!-- Table-specification and miscellaneous marker segment syntax
      -->
    <xs:element name="TableSpecification"
      minOccurs="0" type="TableSpecificationType"/>

    <!-- Frame header -->
    <xs:element name="frameHeader" type="frameHeaderType"/>

    <!-- One or more scan -->
    <xs:element name="scan" type="scanType" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="TableSpecificationType">
  <xs:sequence>
    <!-- Quantization tables marker segment -->
    <xs:element name="DQT" minOccur="0" type="DQTType"/>

    <!-- Huffman tables marker segment -->
    <xs:element name="DHT" minOccur="0" type="DHTType"/>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="DQTType">
  <xs:sequence>
```

```
    <xs:element name="DQTMarker" type="xs:hexBinary"
        fixed="0xffdb"/>
    <xs:element name="lengthOfQT" type="xs:short"/>
    <xs:element name="QTInfo" type="xs:byte">
        <xs:annotation>
           <xs:appinfo>
             <dfdl:dataFormat byteSize="{lengthOfQT-2}"/>
           </xs:appinfo>
        </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="DHTType">
  <xs:sequence>
    <xs:element name="DHTMarker" type="xs:hexBinary"
        fixed="0xffc4"/>
    <xs:element name="lengthOfHT" type="xs:short"/>
    <xs:element name="HTInfo" type="xs:byte">
      <xs:annotation>
        <xs:appinfo>
          <dfdl:dataFormat byteSize="{lengthOfHT-2}"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="frameHeaderType">
  <xs:sequence>
    <xs:element name="SOFMarker" type="xs:hexBinary"
        fixed="0xffc0"/>
    <xs:element name="lengthOfSOF" type="xs:short"/>
    <xs:element name="dataPrecision" type="xs:byte"/>
    <xs:element name="imageHeight" type="xs:short"/>
    <xs:element name="imageWidth" type="xs:short"/>

    <xs:element name="numberOfComponents">
      <xs:simpleType>
        <xs:restriction base="xs:byte">
           <!-- grey scaled -->
          <xs:enumeration value="0x01"/>
          <!-- colour YcbCr(YUV) or YIQ -->
          <xs:enumeration value="0x03"/>
          <xs:enumeration value="0x04"/> <!-- colour CMYK -->
        </xs:restriction>
      </xs:simpleType>
    </xs:element>

    <xs:element name="frameComponent"
      minOccurs="{numberOfComponents}"
      maxOccurs="{numberOfComponents}">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="componentId" type="xs:byte"/>
          <xs:element name="horizontalSamplingFactors"
            type="xs:byte">
            <xs:annotation>
              <xs:appinfo>
                <dfdl:bitstream length="4"/>
              </xs:appinfo>
            </xs:annotation>
          </xs:element>
          <xs:element name="verticalSamplingFactors" type="xs:byte">
            <xs:annotation>
              <xs:appinfo>
                <dfdl:bitstream length="4"/>
              </xs:appinfo>
            </xs:annotation>
          </xs:element>
          <xs:element name="QTDestination" type="xs:byte"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="scanType">
  <xs:sequence>
    <!-- Scan header -->
    <xs:element name="scanHeader" type="scanHeaderType"/>

    <!-- entropy-coded segment -->
    <xs:element name="lastEntropyCodedSegment" type="xs:byte">
      <xs:annotation>
        <xs:appinfo>
          <dfdl:dataFormat byteSize="unbounded"/>
          <dfdl:dataFormat terminator="[0xffda, 0xffc0, 0xffd9]"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>


<xs:complexType name="scanHeaderType">
  <xs:sequence>
    <xs:element name="SOSMarker" type="xs:hexBinary"
        fixed="0xffda"/>
    <xs:element name="lengthOfSOS" type="xs:short"/>

    <xs:element name="numberOfComponents">
      <xs:simpleType>
        <xs:restriction base="xs:byte">
          <!-- grey scaled -->
          <xs:enumeration value="0x01"/>
          <!-- colour YcbCr(YUV) or YIQ -->
          <xs:enumeration value="0x03"/>
          <xs:enumeration value="0x04"/> <!-- colour CMYK -->
        </xs:restriction>
      </xs:simpleType>
    </xs:element>

    <xs:element name="scanComponent"
      minOccurs="{numberOfComponents}"
      maxOccurs="{numberOfComponents}">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="componentId" type="xs:byte"/>
          <xs:element name="DCTDestination" type="xs:byte">
            <xs:annotation>
              <xs:appinfo>
                <dfdl:bitstream length="4"/>
              </xs:appinfo>
            </xs:annotation>
          </xs:element>
          <xs:element name="ACTDestination" type="xs:byte">
            <xs:annotation>
              <xs:appinfo>
                <dfdl:bitstream length="4"/>
              </xs:appinfo>
            </xs:annotation>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="reservedBytes" type="xs:byte">
        <xs:annotation>
            <xs:appinfo>
              <dfdl:dataFormat byteSize="3"/>
            </xs:appinfo>
        </xs:annotation>
    </xs:element>
  </xs:sequence>
 </xs:complexType>

</xs:schema>
```

➢ The XML representation of the original image (JpegXML.xml)

```
<DFDL>
  <JpegDFDL>
    <SOI>ffd8</SOI>
    <APP1>
      <APP1Marker>ffe1</APP1Marker>
      <lengthOfAPP1>3845</lengthOfAPP1>
      <fileId1>Exif</fileId1>
      <fileId2>00</fileId2>
      <fileId3>00</fileId3>
      <byteAlign>4949</byteAlign>
      <tagMark>2a00</tagMark>
      <offsetToIFD0>08000000</offsetToIFD0>
      <IFD0>
        <numberOfEntry>0c00</numberOfEntry>
        <entry>
          <tagNumber>0e01</tagNumber>
          <dataFormat>0200</dataFormat>
          <numberOfEntryComponents>20000000</numberOfEntryComponents>
          <dataValue>9e000000</dataValue>
        </entry>
        <entry>
          <tagNumber>0f01</tagNumber>
          <dataFormat>0200</dataFormat>
          <numberOfEntryComponents>18000000</numberOfEntryComponents>
          <dataValue>be000000</dataValue>
        </entry>
        <entry>
          <tagNumber>1001</tagNumber>
          <dataFormat>0200</dataFormat>
          <numberOfEntryComponents>07000000</numberOfEntryComponents>
          <dataValue>d6000000</dataValue>
        </entry>
        <entry>
          <tagNumber>1201</tagNumber>
          <dataFormat>0300</dataFormat>
          <numberOfEntryComponents>01000000</numberOfEntryComponents>
          <dataValue>01000000</dataValue>
        </entry>
        <entry>
          <tagNumber>1a01</tagNumber>
    <dataFormat>0500</dataFormat>
    <numberOfEntryComponents>01000000</numberOfEntryComponents>
    <dataValue>ee000000</dataValue>
  </entry>
  <entry>
    <tagNumber>1b01</tagNumber>
    <dataFormat>0500</dataFormat>
    <numberOfEntryComponents>01000000</numberOfEntryComponents>
    <dataValue>f6000000</dataValue>
  </entry>
  <entry>
    <tagNumber>2801</tagNumber>
    <dataFormat>0300</dataFormat>
    <numberOfEntryComponents>01000000</numberOfEntryComponents>
    <dataValue>02000000</dataValue>
  </entry>
  <entry>
    <tagNumber>3101</tagNumber>
    <dataFormat>0200</dataFormat>
    <numberOfEntryComponents>09000000</numberOfEntryComponents>
    <dataValue>fe000000</dataValue>
  </entry>
  <entry>
    <tagNumber>3201</tagNumber>
    <dataFormat>0200</dataFormat>
    <numberOfEntryComponents>14000000</numberOfEntryComponents>
    <dataValue>1e010000</dataValue>
  </entry>
  <entry>
    <tagNumber>1302</tagNumber>
    <dataFormat>0300</dataFormat>
    <numberOfEntryComponents>01000000</numberOfEntryComponents>
    <dataValue>02000000</dataValue>
  </entry>
  <entry>
    <tagNumber>6987</tagNumber>
    <dataFormat>0400</dataFormat>
    <numberOfEntryComponents>01000000</numberOfEntryComponents>
    <dataValue>26020000</dataValue>
  </entry>
  <entry>
    <tagNumber>a5c4</tagNumber>
    <dataFormat>0700</dataFormat>
```

```
      <numberOfEntryComponents>04010000</numberOfEntryComponents>
      <dataValue>32010000</dataValue>
    </entry>
    <offsetToIFD1>96040000</offsetToIFD1>
  </IFD0>
  <ignorableBytes>
      4f4c594d505553204449474954414c2043414d455241202020202020202
      020004f4c59 ... ...
  </ignorableBytes>
</APP1>
<frame>
  <TableSpecification>
    <DQT>
      <DQTMarker>ffdb</DQTMarker>
      <lengthOfQT>00c5</lengthOfQT>
      <QTInfo>
          000a07070807060a0808080b0a0a0b0e18100e0d0d0e1d1516111823
          1f2524221f2221 ... ...
      </QTInfo>
    </DQT>
    <DHT>
      <DHTMarker>ffc4</DHTMarker>
      <lengthOfHT>01a2</lengthOfHT>
      <HTInfo>
          0000010501010101010100000000000000000102030405060708090a
          0b100002010303 ... ...
      </HTInfo>
    </DHT>
  </TableSpecification>
  <frameHeader>
    <SOFMarker>ffc0</SOFMarker>
    <lengthOfSOF>0011</lengthOfSOF>
    <dataPrecision>08</dataPrecision>
    <imageHeight>0078</imageHeight>
    <imageWidth>00a0</imageWidth>
    <numberOfComponents>03</numberOfComponents>
    <frameComponent>
      <componentId>01</componentId>
      <horizontalSamplingFactors>02</horizontalSamplingFactors>
      <verticalSamplingFactors>01</verticalSamplingFactors>
      <QTDestination>00</QTDestination>
    </frameComponent>
    <frameComponent>
```

```
          <componentId>02</componentId>
          <horizontalSamplingFactors>01</horizontalSamplingFactors>
          <verticalSamplingFactors>01</verticalSamplingFactors>
          <QTDestination>01</QTDestination>
        </frameComponent>
        <frameComponent>
          <componentId>03</componentId>
          <horizontalSamplingFactors>01</horizontalSamplingFactors>
          <verticalSamplingFactors>01</verticalSamplingFactors>
          <QTDestination>01</QTDestination>
        </frameComponent>
      </frameHeader>
      <scan>
        <scanHeader>
          <SOSMarker>ffda</SOSMarker>
          <lengthOfSOS>000c</lengthOfSOS>
          <numberOfComponents>03</numberOfComponents>
          <scanComponent>
            <componentId>01</componentId>
            <DCTDestination>00</DCTDestination>
            <ACTDestination>00</ACTDestination>
          </scanComponent>
          <scanComponent>
            <componentId>02</componentId>
            <DCTDestination>01</DCTDestination>
            <ACTDestination>01</ACTDestination>
          </scanComponent>
          <scanComponent>
            <componentId>03</componentId>
            <DCTDestination>01</DCTDestination>
            <ACTDestination>01</ACTDestination>
          </scanComponent>
          <reservedBytes>003f00</reservedBytes>
        </scanHeader>
        <lastEntropyCodedSegment>
            f18a29005140094b40094500145001450014500145001450014500145002d1
            4005140094b40 ... ...
        </lastEntropyCodedSegment>
      </scan>
    </frame>
    <EOI>ffd9</EOI>
  </JpegDFDL>
</DFDL>
```

Appendix Ⅳ Java API Document

Here I only listed the Java API document for the core classes of the system, including

Packages	
msc.api	Package of interfaces and abstract classes, including DFDL Schema parser, data file parser, structure item and data item.
msc.bean	Package of all the data related classes, including data beans and different kinds of data items stored in the beans.
msc.impl	Package of the implementation classes for the high-level library, DFDL Schema parser and data file parser.
msc.util	Package of miscellaneous utility classes, including convertor, file access facilities, builders and constants class.

Hierarchy For All Packages

Package Hierarchies:
msc.api, msc.bean, msc.impl, msc.util

Class Hierarchy

- class java.lang.**Object**
  - class msc.util.**Convertor**
  - class msc.bean.**DataBean**
  - class msc.util.**DataBeanBuilder**
  - class msc.api.**DataItem**
    - class msc.bean.**LabelDataItem**
    - class msc.bean.**SimpleDataItem**
  - class msc.util.**DataParserUtilities**
  - class msc.impl.**DFDLLibrary**
  - class msc.util.**Directives**
  - class msc.util.**ErrorMessage**
  - class msc.util.**FileWrapper**
  - class msc.bean.**LookupTable**
  - class msc.util.**PrimitiveType**
  - class msc.util.**Printer**
  - class msc.util.**SchemaUtilities**
  - class msc.impl.**SimpleDataFileParser** (implements msc.api.DataFileParser)
  - class msc.impl.**SimpleDFDLParser** (implements msc.api.DFDLParser)
  - class msc.bean.**StructureBean**
  - class msc.util.**StructureBeanBuilder**
  - class msc.api.**StructureItem**
    - class msc.bean.**ComplexStructureItem**
    - class msc.bean.**LabelStructureItem**
    - class msc.bean.**SimpleStructureItem**
  - class msc.util.**XMLBuilder**

Interface Hierarchy

- interface msc.api.**DataFileParser**
- interface msc.api.**DFDLParser**

msc.util

Class Convertor

```
java.lang.Object
  └msc.util.Convertor
```

public class **Convertor**
extends Object

The class containing different kinds of useful translation and conversion methods

Author:
Yi Zhu

Constructor Summary

`Convertor()`

Method Summary

`static String`	`bytesToHexString(byte[] bytes)` Returns a string representation of a bytes array in base 16
`static int`	`bytesToInt(byte[] bytes, String byteOrder)` Convert a bytes array to the corresponding int value based on a particular byte order
`static String`	`bytesToText(byte[] bytes)` Return the text representaion of a bytes array based on the Unicode character
`static double`	`expToValue(String exp, LookupTable varTable)` Evaluate the value of the formular expression containing both figures and variables
`static String`	`formatString(String s)` Format the string, deleting the "{", "}" and space
`static ArrayList`	`getTerminator(String terminator)` Extract terminators from a terminator expression like "[0xff,

	0xd9, 0x34]"
static ArrayList	getVarFromExp(String exp) Extract variables from a expression containing both figures and variables

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Convertor

```
public Convertor()
```

Method Detail

bytesToHexString

```
public static String bytesToHexString(byte[] bytes)
```

Returns a string representation of a bytes array in base 16

Parameters:
`bytes` - the bytes array to be converted

Returns:
the hexadecimal string representation of the bytes array

---

bytesToInt

```
public static int bytesToInt(byte[] bytes,
                             String byteOrder)
```

Convert a bytes array to the corresponding int value based on a particular byte order

Parameters:
`bytes` - the original bytes array
`byteOrder` - the byte order attribute

Returns:
the int value of the bytes array

---

bytesToText

```
public static String bytesToText(byte[] bytes)
```

Return the text representaion of a bytes array based on the Unicode character

Parameters:
`bytes` - the bytes array to be converted

Returns:
the text representaion of the bytes array

---

expToValue

```
public static double expToValue(String exp,
                                LookupTable varTable)
```

Evaluate the value of the formular expression containing both figures and variables

Parameters:
`exp` - the expression to be evaluated
`varTable` - the table containing all variables and their values

Returns:
the value of the expression

---

formatString

```
public static String formatString(String s)
```

Format the string, deleting the "{", "}" and space

Parameters:
`s` - the original string

Returns:
a new string containing all characters of the original string except "{", "}" and space

---

getTerminator

```
public static ArrayList getTerminator(String terminator)
```

Extract terminators from a terminator expression like "[0xff, 0xd9, 0x34]"

Parameters:

`terminator` - the original terminator expression

Returns:

an array of the terminators contained in the original terminator expression

---

getVarFromExp

```
public static ArrayList getVarFromExp(String exp)
```

Extract variables from a expression containing both figures and variables

Parameters:

`exp` - the original expression

Returns:

an array of variables in the expression

---

msc.impl

Class DFDLLibrary

```
java.lang.Object
  └msc.impl.DFDLLibrary
```

public class **DFDLLibrary**
extends Object

The high-level library encapsulating all required functionalities

Author:
Yi Zhu

Constructor Summary
`DFDLLibrary()`

Method Summary	
`void`	`generateXMLRepresentation(String xmlURI)` Generate the cooresponding XML file which represents the data file
`DataBean`	`getDataBean()` Return the data bean containing the actual data.
`StructureBean`	`getStructureBean()` Return the structure bean containing the structure information.
`void`	`parseDataFile(String dataFileURI)` An interface for expert clients, parse the data file according to the structure information from the `StructureBean`, setting the `DataBean`
`void`	`parseSchema(String DFDLSchemaURI)` An interface for expert clients, parse the DFDL schema, setting the `StructureBean`
`void`	`simpleParse(String DFDLSchemaURI, String dataFileURI)` A simple interface for un-expert clients, simply parsing the

	DFDL schema and an data file at the same time, setting the `StructureBean` and the `DataBean`

Methods inherited from class java.lang.Object
`equals`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor Detail

DFDLLibrary

`public` **`DFDLLibrary`**`()`

Method Detail

generateXMLRepresentation

`public void` **`generateXMLRepresentation`**`(String xmlURI)`

Generate the cooresponding XML file which represents the data file

Parameters:
`xmlURI` - the URI giving the location of the output XML representation

---

getDataBean

`public DataBean` **`getDataBean`**`()`

Return the data bean containing the actual data.

Returns:
the `DataBean` containing the data

---

getStructureBean

`public StructureBean` **`getStructureBean`**`()`

Return the structure bean containing the structure information.

Returns:
the `StructureBean` containing the structure information

parseDataFile

```
public void parseDataFile(String dataFileURI)
```

An interface for expert clients, parse the data file according to the structure information from the `StructureBean`, setting the `DataBean`

Parameters:
`dataFileURI` - the URI giving the base location of the original data file

parseSchema

```
public void parseSchema(String DFDLSchemaURI)
```

An interface for expert clients, parse the DFDL schema, setting the `StructureBean`

Parameters:
`DFDLSchemaURI` - the URI giving the base location of the DFDL Schema

simpleParse

```
public void simpleParse(String DFDLSchemaURI,
                        String dataFileURI)
```

A simple interface for un-expert clients, simply parsing the DFDL schema and an data file at the same time, setting the `StructureBean` and the `DataBean`

Parameters:
`DFDLSchemaURI` - the URI giving the base location of the DFDL Schema
`dataFileURI` - the URI giving the base location of the original data file

msc.util

Class DataBeanBuilder

```
java.lang.Object
  └msc.util.DataBeanBuilder
```

public class **DataBeanBuilder**
extends Object

The builder for building a date bean step by step

Author:
Yi Zhu

Constructor Summary

`DataBeanBuilder()` Constructs a `DataBeanBuilder`

Method Summary

`void`	`buildItem(NodeTree subTree)` Insert sub tree at the root
`void`	`buildRoot(Object object)` Build the root element
`DataBean`	`getDataBean()` Return the result data bean
`void`	`mergeTree(NodeTree parentTree, Position parent, NodeTree subTree)` Merge the children of sub-tree to the parent node of the parent tree

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DataBeanBuilder

```
public DataBeanBuilder()
```

Constructs a `DataBeanBuilder`

Method Detail

buildItem

```
public void buildItem(NodeTree subTree)
```

Insert sub tree at the root

---

buildRoot

```
public void buildRoot(Object object)
```

Build the root element

Parameters:
`object` - the object to be built

---

getDataBean

```
public DataBean getDataBean()
```

Return the result data bean

Returns:
the result DataBean

---

mergeTree

```
public void mergeTree(NodeTree parentTree,
                      Position parent,
                      NodeTree subTree)
```

Merge the children of sub-tree to the parent node of the parent tree

Parameters:

`parentTree` - the parent tree
`parent` - the parent node to which the children of sub-tree will be added
`subTree` - whose children to be merged to the parent tree

---

msc.util

Class DataParserUtilities

```
java.lang.Object
  └─msc.util.DataParserUtilities
```

public class **DataParserUtilities**
extends Object

Provide utility methods for processing the DFDL Schema

Author:
Yi Zhu

Constructor Summary
`DataParserUtilities(FileWrapper dataFile, LookupTable lookupTable, HashMap simpleTypes, HashMap complexTypes)` Constructs a new `DataParserUtilities` instance with necessary proporties

Method Summary	
`SimpleStructureItem`	`findSimpleItem(Object nextItem)` Find the first simple data item from the specified data item
`boolean`	`isOccur(Object object)` Return true if the specified data item occurs, otherwise return false
`int`	`parseByteSize(String byteSize)` Translate the string value of "byteSize" attribute into actual value

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DataParserUtilities

```
public DataParserUtilities(FileWrapper dataFile,
                           LookupTable lookupTable,
                           HashMap simpleTypes,
                           HashMap complexTypes)
```

Constructs a new `DataParserUtilities` instance with necessary proporties

Parameters:

`dataFile` - the data file being read
`lookupTable` - the lookup table from which the contents of previous data elements can be get
`simpleTypes` - the mapping between all simple type attributes and their names
`complexTypes` - the mapping between all complex type structure and their names

Method Detail

findSimpleItem

```
public SimpleStructureItem findSimpleItem(Object nextItem)
```

Find the first simple data item from the specified data item

Parameters:
`nextItem` - the data item to be searched
Returns:
the first `SimpleStructureItem` contained in this data item

---

isOccur

```
public boolean isOccur(Object object)
```

Return true if the specified data item occurs, otherwise return false

Parameters:
`object` - the data item to be tested
Returns:
Return true if the specified data item occurs, otherwise return false

---

parseByteSize

```
public int parseByteSize(String byteSize)
```

Translate the string value of "byteSize" attribute into actual value

Parameters:

`byteSize` - the string representation of the "byteSize" attribute

Returns:

the actual value of the "byteSize" attribute msc.util

Class FileWrapper

```
java.lang.Object
  └msc.util.FileWrapper
```

public class **FileWrapper**
extends Object

A file wrapper class providing more convenient access to the file, including different kinds of read method

Author:
Yi Zhu

Constructor Summary

`FileWrapper(String URI)` Constructor of the FileWrapper class, instancing the InputStream of the specified URI

Method Summary

`File`	`file()` Return the file object
`int`	`length()` Return the length of the file
`boolean`	`match(byte[] data, ArrayList enum)` Check whether the bytes array equals to any one of the terminator stored in the enumeration list
`boolean`	`match(byte[] data, String terminator)` Check whether the bytes array equals to the terminator
`int`	`offset()` Return the offset of the file
`byte[]`	`readByte(int n)` Read n bytes of data from the input stream into an array of bytes,

	and then return the bytes array
`byte[]`	`readUntil(ArrayList enum, int byteSize)` Read data from the input stream into an array of bytes until arriving at any one of the terminators stored in the enumeration list, and then return the bytes array
`byte[]`	`readUntil(String terminator, int byteSize)` Read data from the input stream into an array of bytes until arriving at the terminator, and then return the bytes array

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

FileWrapper

`public FileWrapper(String URI)`

Constructor of the FileWrapper class, instancing the InputStream of the specified URI

Parameters:
`URI` -

Method Detail

file

`public File file()`

Return the file object

Returns:
the file object

---

length

`public int length()`

Return the length of the file

Returns:
the length of the file

---

match

```
public boolean match(byte[] data,
                     ArrayList enum)
```

Check whether the bytes array equals to any one of the terminator stored in the enumeration list

Parameters:
`data` - the bytes array to be checked
`enum` - the enumeration list containing all possible terminators
Returns:
true if the bytes array equals to any one of the terminator stored in the enumeration list; false otherwise

---

match

```
public boolean match(byte[] data,
                     String terminator)
```

Check whether the bytes array equals to the terminator

Parameters:
`data` - the bytes array to be checked
`terminator` - the terminator to be cheched
Returns:
true if they are same; false otherwise

---

offset

```
public int offset()
```

Return the offset of the file

Returns:
the offset of the file

---

readByte

```
public byte[] readByte(int n)
```

Read n bytes of data from the input stream into an array of bytes, and then return the bytes array

Parameters:
`n` - the number of bytes to be read

Returns:
the bytes array containing the data

---

readUntil

```
public byte[] readUntil(ArrayList enum,
                        int byteSize)
```

Read data from the input stream into an array of bytes until arriving at any one of the terminators stored in the enumeration list, and then return the bytes array

Parameters:
`enum` - the enumeration list containing all possible terminators
`byteSize` - the byte size of the terminator

Returns:
the bytes array containing the data

---

readUntil

```
public byte[] readUntil(String terminator,
                        int byteSize)
```

Read data from the input stream into an array of bytes until arriving at the terminator, and then return the bytes array

Parameters:
`terminator` - the terminator for stoping read
`byteSize` - the byte size of the terminator

Returns:
the bytes array containing the data

---

msc.util

Class SchemaUtilities

```
java.lang.Object
  └msc.util.SchemaUtilities
```

public class **SchemaUtilities**
extends Object

The class containing utilities methods for processing XML Schema

Author:
Yi Zhu

Constructor Summary

`SchemaUtilities()`

Method Summary

`Element`	`getChildByTagName(Element parent, String tagName)` Return the first child element of the parent element, which has the specified tag name
`Element`	`getElementByAttr(Element parent, String attr, String value)` Return the first child element of the parent, which has the specified attribute that has the specified value
`Element`	`getFirstChildElement(Element parent)` Return the first child element of the parent

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

SchemaUtilities

```
public SchemaUtilities()
```

Method Detail

getChildByTagName

```
public Element getChildByTagName(Element parent,
                                 String tagName)
```

Return the first child element of the parent element, which has the specified tag name

Parameters:
`parent` - the parent element whose children will be examined
`tagName` - the requested tag name

Returns:
the child element with the specified tag name; return `NULL` if not found

---

getElementByAttr

```
public Element getElementByAttr(Element parent,
                                String attr,
                                String value)
```

Return the first child element of the parent, which has the specified attribute that has the specified value

Parameters:
`parent` - the parent element whose children will be examined
`attr` - the requested attribute
`value` - the requested value of the requested attribute

Returns:
the child element with the specified attribute which has the specified value; return `NULL` if not found

---

getFirstChildElement

```
public Element getFirstChildElement(Element parent)
```

Return the first child element of the parent

Parameters:

`parent` - the parent element from which the child will be found

Returns:

the first child element of the parent

msc.impl

Class SimpleDFDLParser

```
java.lang.Object
  └msc.impl.SimpleDFDLParser
```

All Implemented Interfaces:
DFDLParser

---

public class **SimpleDFDLParser**
extends Object
implements DFDLParser

The DFDL Schema parser, parsing structure information, which is required by parsing data files, from a DFDL descrption

Author:
Yi Zhu

Constructor Summary
`SimpleDFDLParser(String schemaURI)` The constructor, setting the base location of the DFDL Schema.

Method Summary	
`StructureBean`	`parseDFDLSchema()` Parse the DFDL schema and the associated data file, generating the result structureBean containing the strucuture information.

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

SimpleDFDLParser

```
public SimpleDFDLParser(String schemaURI)
```

The constructor, setting the base location of the DFDL Schema.

Parameters:

`schemaURI` - the URI giving the base location of the DFDL Schema

Method Detail

parseDFDLSchema

```
public StructureBean parseDFDLSchema()
```

Parse the DFDL schema and the associated data file, generating the result structureBean containing the strucuture information.

Specified by:

`parseDFDLSchema` in interface `DFDLParser`

Returns:

the result structureBean

msc.impl

Class SimpleDataFileParser

```
java.lang.Object
  └msc.impl.SimpleDataFileParser
```

All Implemented Interfaces:
DataFileParser

---

public class **SimpleDataFileParser**
extends Object
implements DataFileParser

The data file parser class for parsing the data file based on a structure bean resulted from a SimpleDFDLParser

Author:
Yi Zhu

Constructor Summary
`SimpleDataFileParser(StructureBean structureBean)` The constructor, setting the structure bean from which all required structure information comes

Method Summary	
`DataBean`	`parseDataFile(String dataFileURI)` Parse the data file, according to the structure information provided by the structure bean

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

SimpleDataFileParser

`public SimpleDataFileParser(StructureBean structureBean)`

The constructor, setting the structure bean from which all required structure information comes

Parameters:
`structureBean` - the `StructureBean`

Method Detail

parseDataFile

`public DataBean parseDataFile(String dataFileURI)`

Parse the data file, according to the structure information provided by the structure bean

Specified by:
`parseDataFile` in interface `DataFileParser`

Parameters:
`dataFileURI` - the location of the data file to be parsed

Returns:
the result `DataBean` containing all the data in the data file, according to the structure information from the `StructureBean`

msc.util

Class StructureBeanBuilder

```
java.lang.Object
  └msc.util.StructureBeanBuilder
```

public class **StructureBeanBuilder**
extends Object

The builder for building a structure bean step by step

Author:
Yi Zhu

Constructor Summary

StructureBeanBuilder()   Constructs a new `StructureBeanBuilder`

Method Summary

`void`	**addAttribute**(Object key, Object value) Add a global attribute to the attributes `HashMap` of the `StructureBean`
`void`	**addComplexType**(Object key, Object value) Add a complex type to the complex type `HashMap` of the `StructureBean`
`void`	**addSimpleType**(Object key, Object value) Add a simple type to the simple type `HashMap` of the `StructureBean`
`void`	**buildRoot**(Object object) Build the root element
`void`	**buildTree**(NodeTree subTree) Add a sub-tree to the structure tree of the `StructureBean`
`StructureBean`	**getStructureBean**() Return the result `StructureBean`

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

StructureBeanBuilder

```
public StructureBeanBuilder()
```

Constructs a new `StructureBeanBuilder`

Method Detail

addAttribute

```
public void addAttribute(Object key,
                         Object value)
```

Add a global attribute to the attributes `HashMap` of the `StructureBean`

Parameters:
`key` - the name of the attribute to be added
`value` - the value of the attribute

---

addComplexType

```
public void addComplexType(Object key,
                           Object value)
```

Add a complex type to the complex type `HashMap` of the `StructureBean`

Parameters:
`key` - the name of the complex type
`value` - the structure information of the complex type

---

addSimpleType

```
public void addSimpleType(Object key,
                          Object value)
```

Add a simple type to the simple type `HashMap` of the `StructureBean`

Parameters:
`key` - the name of the simple type
`value` - the structure information of the simple type

---

buildRoot

```
public void buildRoot(Object object)
```

Build the root element

Parameters:
`object` - the object to be built

---

buildTree

```
public void buildTree(NodeTree subTree)
```

Add a sub-tree to the structure tree of the `StructureBean`

Parameters:
`subTree` - the sub-tree to be added

---

getStructureBean

```
public StructureBean getStructureBean()
```

Return the result `StructureBean`

Returns:
the result StructureBean

---

msc.util

Class XMLBuilder

```
java.lang.Object
  └msc.util.XMLBuilder
```

public class **XMLBuilder**
extends Object

The builder for building the XML representation of a data file step by step

Author:
Yi Zhu

Constructor Summary

`XMLBuilder``(String xmlURI)` Constructs a new `XMLBuilder`, specifying the target output file

Method Summary

`void`	**`buildCloseLabel`**`(LabelDataItem item, Integer rank)` Build a close XML tag, which is "/" with a label, for the specified `LabelDataItem`
`void`	**`buildComplexDataItem`**`(NodeTree tree, Position pos, Integer rank)` Represent a complex type data item in XML representation
`void`	**`buildLabel`**`(LabelDataItem item, Integer rank)` Build a XML tag, which is a label, for the specified `LabelDataItem`
`void`	**`buildSimpleDataItem`**`(Position pos, Integer rank)` Represent a simple type data item in XML representation
`void`	**`close`**`()` Close the file output stream and releases any system resources

Methods inherited from class java.lang.Object
`equals, getClass, hashCode, notify, notifyAll, toString, wait,`

`wait, wait`

Constructor Detail

XMLBuilder

```
public XMLBuilder(String xmlURI)
```

Constructs a new `XMLBuilder`, specifying the target output file

Parameters:
`xmlURI` - the location and file name of the result XML representation output

Method Detail

buildCloseLabel

```
public void buildCloseLabel(LabelDataItem item,
                            Integer rank)
```

Build a close XML tag, which is "/" with a label, for the specified `LabelDataItem`

Parameters:
`item` - the `LabelDataItem` to be built
`rank` - its rank in the XML file

---

buildComplexDataItem

```
public void buildComplexDataItem(NodeTree tree,
                                 Position pos,
                                 Integer rank)
```

Represent a complex type data item in XML representation

Parameters:
`tree` - the sub-tree containing all child data elements of the complex type data item to be represented
`pos` - the position of the complex type data item to be represented
`rank` - its rank in the XML file

---

buildLabel

```
public void buildLabel(LabelDataItem item,
                       Integer rank)
```

Build a XML tag, which is a label, for the specified `LabelDataItem`

Parameters:
`item` - the `LabelDataItem` to be built
`rank` - its rank in the XML file

---

buildSimpleDataItem

```
public void buildSimpleDataItem(Position pos,
                                Integer rank)
```

Represent a simple type data item in XML representation

Parameters:
`pos` - the position of the simple type data item to be represented
`rank` - its rank in the XML file

---

close

```
public void close()
```

Close the file output stream and releases any system resources

---
