Tutoriel DCM4CHE3

Par Dimitri PIANETA

Version 2016

Table des matières

I) Historique :

Dans les années 2000, Gunter ZEILINGER avait écrit JDicomutility suite utilisé par le Soflink (maintenant TriSpark) Java DICOM Toolkit (JDT). Après cette expérience avec le JDT, il décidait décrire un package DICOM toolkit. Ainsi est née dcm4che (prononcé d-c-m-for-chay).

Le site internet est le suivant : www.dcm4che.com

II) Comment l'installer :

1. Télécharger le package dcm4che3 à l'adresse suivante
https://sourceforge.net/projects/dcm4che/files/dcm4che3/
2. Installer le JDK 8 (version minimal 7).

Choix selon OS:
MS windows seven (64 bits) --> JRE(64bits) ou JDK(32bits)
MS windows XP (32bits) --> JRE(32bits) ou JDK(32bits)
MS windows XP (64bits) --> JRE(64bits) ou JDK(64bits)

3. Inclure dans l'EDI que vous utilisez les librairies dcm4che 3(

4. Installer jai_imageio pour 32 bits.

Mais pour jai_imageio pour OS windows jdk 7, ne fonctionne pas car problèmes de bibliothèque qui n'est pas maintenu.

Copier à la racine du répertoire JDK l'exécutable jai_imageio-1_1-alpha-lib-windows-i586-jdk.exe ou Java Adavanced Imaging I/o Tool 1.0_01 sur url: http://java.sun.com/products/java-media/jai/downloads/download-iio-1_0_01.html
ou http://download.java.net/media/jai-imageio/builds/release/1.1/ parce que nous utilisons du JPEG2000.

III) Comment utiliser Dcm4che3:

III.1) Gestion de la liste des tags en DICOM pour un fichier standard (une seule image dans le fichier) :

a) But :

On souhaite afficher les tags.

b) Entrée en java Standard :

Les bibliothèques d'E/S utilisent souvent l'abstraction d'un flux [stream], qui représente n'importe quelle source ou réceptacle de données comme un objet capable de produire et de recevoir des parties de données. Le flux cache les détails de ce qui arrive aux données dans le véritable dispositif d'E/S.

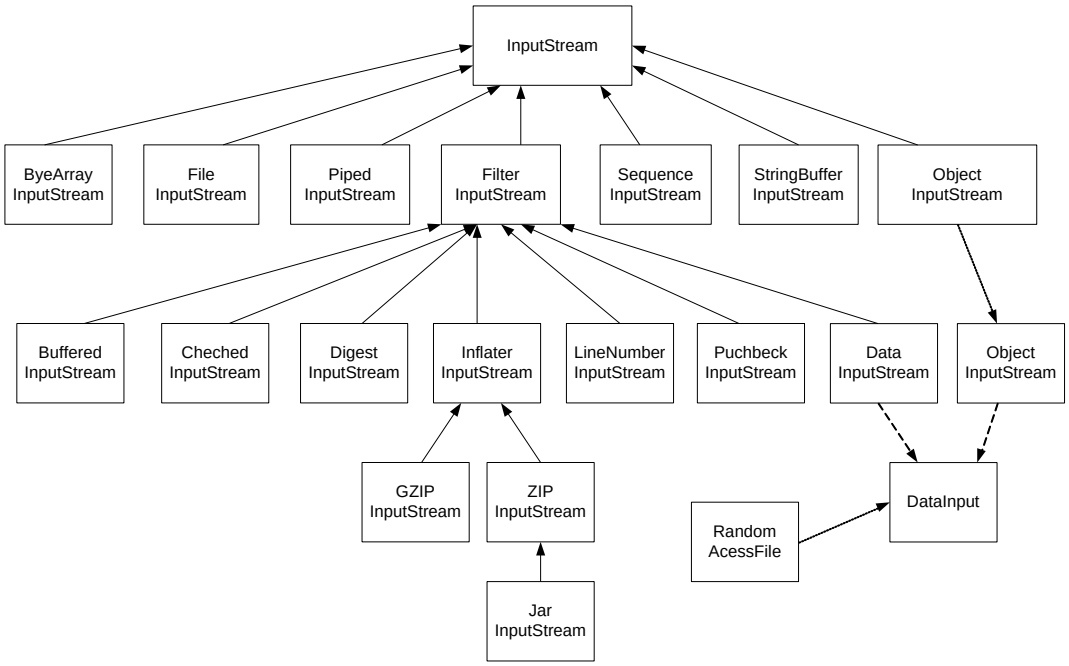


Figure 1 : Schéma de l'entrée

Sous-classe	Fonction
FileInputStream	Permet de créer un flux avec un fichier présent dans le système de fichiers. Cette classe possède un constructeur prenant en paramètre un objet de type File ou un String , qui représente le chemin vers le fichier.
ByteArrayInputStream	Permet de lire des données binaires à partir d'un tableau d'octets.
PipedInputStream	Permet de créer une sorte de tube d'entrée (pipe). Dans celui-ci, les informations circuleront sous forme d'octets. Cette classe possède un constructeur ayant pour paramètre un objet de type PipedOutputStream. On peut ainsi connecter les deux tubes ; en gros, ce qui est écrit dans une extrémité peut être lu depuis l'autre.
BufferedInputStream	Cette classe permet la lecture de données à l'aide d'un tampon, un buffer si vous préférez. À l'instanciation, un tableau d'octets est créé afin de servir de tampon et permet de ne pas surcharger la mémoire. Ce tableau est redimensionné automatiquement à chaque lecture pour contenir les données provenant du flux d'entrée. Ce type d'objet est particulièrement approprié lors de traitement de fichiers volumineux !
DataInputStream	Cet objet sert à lire des données représentant des types primitifs de Java (int, boolean, double, byte, ...) préalablement écrits par un DataOutputStream. Grâce à cet objet, vous pouvez récupérer des éléments sérialisés avec des méthodes comme readInt(), readDouble()...
PushbackInputStream	Lit un flux binaire en entrée et replace le dernier octet lu dans le flux !
LineNumberInputStream	Permet d'avoir les numéros de lignes lues dans le flux en plus de lire le flux lui-même.
SequenceInputStream	Permet de concaténer deux (ou plus) flux d'entrée, ce qui permet de traiter plusieurs flux d'entrée comme un seul et unique flux !
ObjectInputStream	Permet de «désérialiser» un objet, c'est-à-dire de restaurer un objet préalablement sauvegardé à l'aide d'un ObjectOutputStream. Cet objet est l'homologue de l'objet DataInputStream, à la différence que celui-ci traite des objets.

Tableau 1 : Rappel de la gestion d'ouverture d'un fichier en java

c) Lire/Écrire dans des fichiers

Le principe d'ouverture d'un fichier est très simple. On fait une lecture d'octet de donnée donc un flux d'information.

FileInputStream : dérivée d'*InputStream* permet d'effectuer une lecture d'octets dans un fichier.

On associe à un objet de type *File,* le flux correspond alors à un flux de fichier :

```
File f =new File(« nomDuFichier »)

FileInputStream f1 = new FileInputStream(f) ;
```

La lecture d'un octet se fait alors avec la méthode *read* :

```
Byte b = f1.read()
```

d) Lire/Écrire des données numériques :

La classe *DataInputStream* permet de lire des données de type numériques tels que les flottants, les booléens, les octets ou les entiers (type double, booléen, byte et int).

Méthodes de *DataInputStream* (Non exhaustif)	
int	`read(byte[] b)` Lit l'ensemble des octets du flux et les stocke dans un tableau. Renvoie –1 si la fin du flux a été atteinte.
int	`read(byte[] b, int off, int len)` Lit le flux à partir de *off* sur *len* octets. Renvoie –1 si la fin du flux est atteinte
boolean	`readBoolean()` Renvoie un booléen égal à *true* si l'octet lu est non nul et *false* si l'octet est nul
byte	`readByte()` Renvoie l'octet lu.
char	`readChar()` Renvoie le caractère correspondant à l'octet lu
double	`readDouble()` Lit 8 octets et renvoie le double correspondant.

e) Chainage des flux filtrés :

```
FileInputStream fis = new FileInputStream("nomFichier" );

DataInputStream dis = new DataInputStream(fis);
```

Cet enchainement va vous permettre d'associer les différents types de flux de manière à bénéficier de leurs fonctionnalités. En effet, partons du principe que *DataInputStream* sait lire des numériques et *FileInputStream* sait lire dans des fichiers et associons les pour qu'ils nous fassent partager leurs compétences propres.

```
FileInputStream dis = new FileInputStream(new BufferedInputStream
        (new FileInputStream("fiche.dat")));
```

L'objet de type DataInputStream est haut de la chaîne de flux puisque ses méthodes d'accès aux données nous serons utiles pour lire des données de type numériques. La méthode read() de l'objet « dis » va procéder à une lecture bufférisée dans le flux.

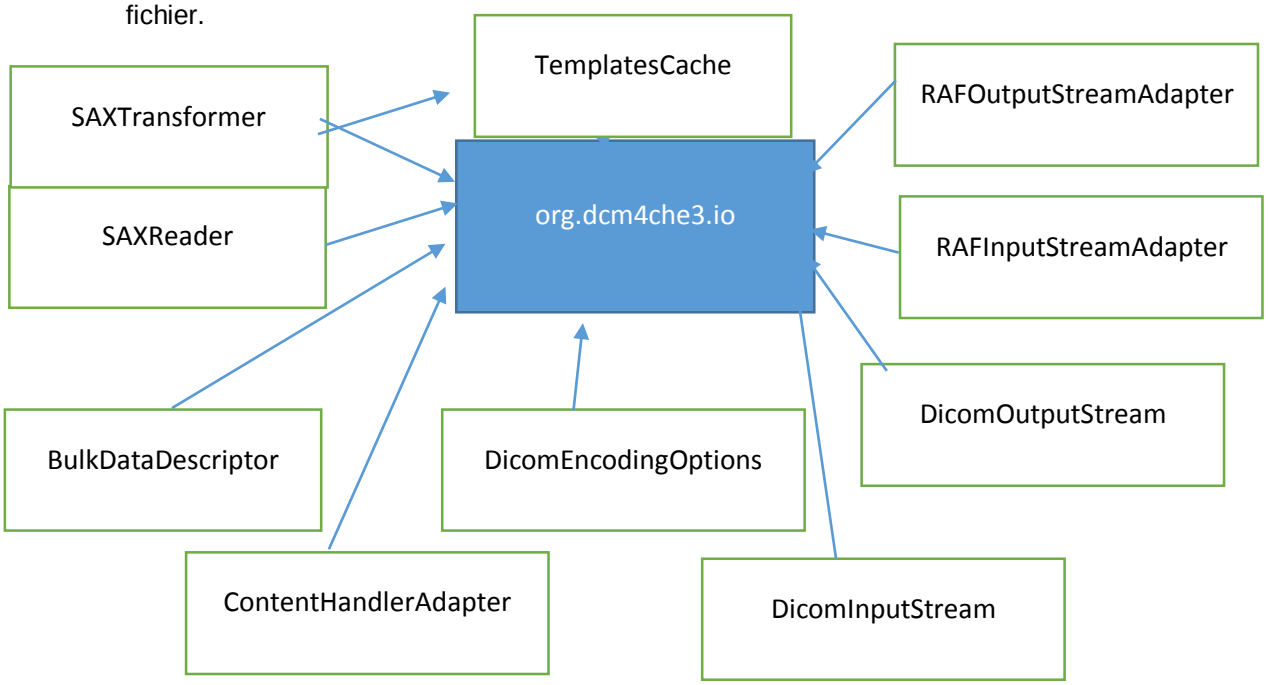
f) Lecture dans un fichier

Cet exemple d'ouverture d'un fichier .txt.

```
String fileString = "";

FileReader fr;

try {

        fr = new FileReader("data.txt");

        int i=0;

        while ((I = fr.read()) !=-1;

        fr.close();
```

g) Lecture dans un fichier Dicom

On va se rappeler par un petit graphe les différentes structures de ce package pour l'ouverture d'un fichier.



Pour ouvrir une image en dicom, nous avons cette suite instruction suivante :

- Filtrer le fichier entrée en vérifiant que c'est un fichier dicom : avec DicomInputStream(File file)
- On lit les tags dans le fichier donc les métadatas dicom avec Attributes

```
DicomInputStream dis = new DicomInputStream(new File("im2.dcm"));

Attributes object =  dis.readDataset(-1, -1);
```

Mais pour trouver les données DICOM rangé dans une image, on doit parcourir chaque ligne du fichier.

Pour comprendre le phénomène de lecture des métadonnées, nous devons faire un petit rappel sur le standard DICOM.

Rappel :

Un fichier Dicom est composé d'une image (les pixels de l'image) qui se trouve dans un champs DICOMs + des méta-données(les tags).

Les métadonnées sont rangées comme une file d'attente. Les tags sont toujours rangés de la même façon.

Le standard DICOM est basé sur un encodage binaire de ces métadonnées sur les normes ISO actuelles de l'imagerie (JPEG, JPEG2000). On dit métadonnées, les informations qui donnent des renseignements de l'image.

Nous allons dans un premier temps voir la structure de l'encodage binaire des métadonnées du standard DICOM. On prend l'exemple classique de David CLUNIE (fondateur de Dicom3) :

Nous souhaitons par exemple trouver la colonne (donc la hauteur de l'image DICOM)

On définit **TAG** (étiquette) **(fig2)** est une paire ordonnée de 16 bits d'entiers non-signés représentant le numéro du groupe suivi par le numéro de l'élément. Le tag est représenté par l'attribut « **group** » qui représente un groupe d'élément de donnée caractéristique. On peut faire analogie avec le langage JAVA c'est la classe par exemple la class patient.

Puis le deuxième élément « **element** » est son les caractéristiques physiques, donnée spécifique de la classe. Reprenons notre exemple de class patient, on pourrait avoir le nom du patient, son âge, son adresse…

Le **VR** signifie la Value Representation. Le VR est une chaîne de deux caractères qui est codée en 2 octets associée à une certaine valeur du Data Tag Element. Chaque TAG doit avoir un VR qui est fixé par le standard DICOM. Les VR sont définies dans le standard DICOM dans la partie 6 nommée le Data Dictionary.

Le **Length** est défini suivant deux ensembles :

- soit un entier non-signé de 16 ou 32 bits dépendant d'un VR explicite ou implicite, et contenant la longueur explicite du champ de valeur en nombre d'octets.
- soit un ensemble de champs de longueur 32 bits. Les longueurs indéfinies pourraient être utilisées pour les éléments de données ayant le VR égal à SQ (Sequence of items) et UN (Unknown ou autres)

Le **Value Field** est un champ de valeur et de longueur variable. Il correspond à l'information identifié par le premier champ. Le type de données de valeurs enregistré dans ce champ est spécifié par le VR de l'élément de données. Par exemple si VR a pour normalisation CS (Code String) alors dans le champs value nous devons avoir une chaîne de caractère, si VR est IS (Integer String) alors dans le champ nous aurons un entier (vue par l'utilisateur). Le champ value est l'image des caractères ASCII ce qui signifie que value est une valeur codée en binaire.

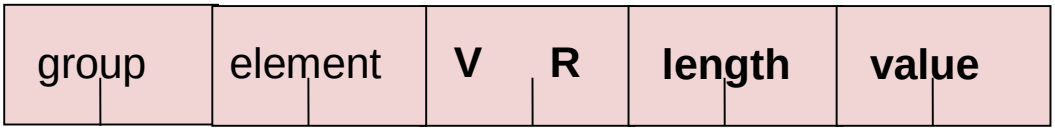


Figure 2: définit le fonctionnement d'une ligne métadonnée

Par ce dernier point, on vient de voir que la spécificité du standard DICOM n'est pas facile par rapport à la représentation actuelle XML, PHP. Alors ce type de codage pour DICOM n'est pas facile avec l'encryptage en binaire.

Prenons comme exemple pour voir le codage binaire :

Tag (0028, 0010) qui représente le nombre de lignes de l'image. Le champ de longueur est représenté en binaire non-signé (Unsigned Short) sur deux octets. On aura par exemple une valeur totale de ligne dans cette image de 256.

28	00	10	00	55	53	02	00	00	01

Pour notre exemple (0028,0010) à pour VR = US qui est égale à une longueur de 2 octets (2*8bits).

On peut résumer notre analyse par cette figure qui représente le nombre de bits.

On peut voir que le tag est représenté par le regroupement du group et element qui sont chacun codés sur 16 bits (2 octets). Puis de la VR (Value Representation) qui est de 0, ou 16 ou 32 bits.

On poursuit cette séquence par la longueur de la « value » qui peut prendre soit 16 ou 32 bits.

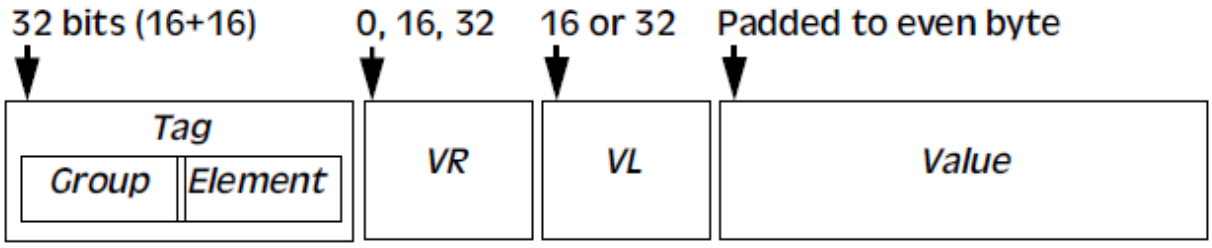
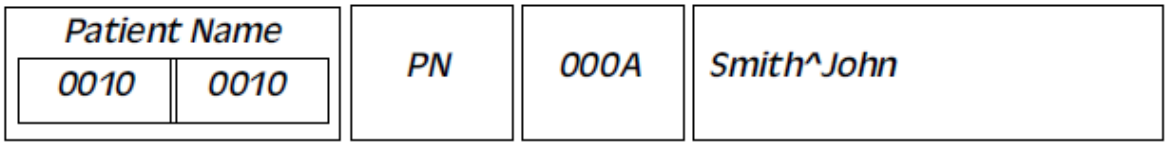


Figure 3: représente nombre bits selon le champ voulu.

Autre exemple pour bien comprendre :



On a tag (0010, 0010) qui signifie que s'est le nom du patient, un VR = PN (Person Name) qui a une longueur de 64 caractères par groupe.

Si on traduit cette ligne comme la machine comprend donc en code hexadécimale[1] :

00 10 00 10 50 49 00 00 00 0A 53 6D 69 74 68 5E 4A 6F 68 6E

[1] Voir annexe : code ASCII

VR Length Value

L'explication de VL est la suivante extraite du livre David Clunie anglais :

The number of bits used to encode the VR and VL fields depends on the transfer syntax and the VR. In the default implicit VR little endian transfer syntax, the VR is never sent and the VL is always 32 bits. In the all other transfer syntaxes, the VR is always sent and is 16 bits. For OB, OW, SQ, UT and UN VRs, the 16 bit VR is followed by 16 bits of zeroes, then a 32 bit VL. For all other VRs, a 16 bit VR field is followed by a 16 bit VL. The idea is to stay aligned on a 32 bit boundary when 32 bit VLs are required for "big" values, and save 32 bits for the others. In retrospect, this "optimization" has caused nothing but trouble, particularly when adding new VRs to the standard, or trying to send "big" values in "small" VRs.

On va voir maintenant : comment les métadonnées sont rangés dans un fichier de type DICOM. On sait que les métadonnées sont codées par-dessus un standard d'image classique comme JPEG-2000.

Les éléments de données sont ensuite ordonnés par ordre croissant pour former un dataset, corps du message. On peut dire en gros que s'est une encapsulation des entêtes comme une file d'attente.

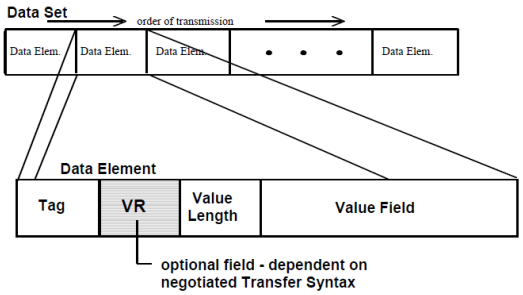
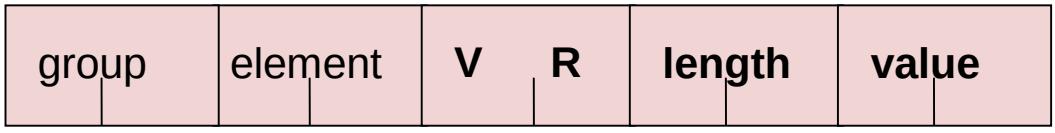


figure 3 : explication la façon d'encodage des métadonnées

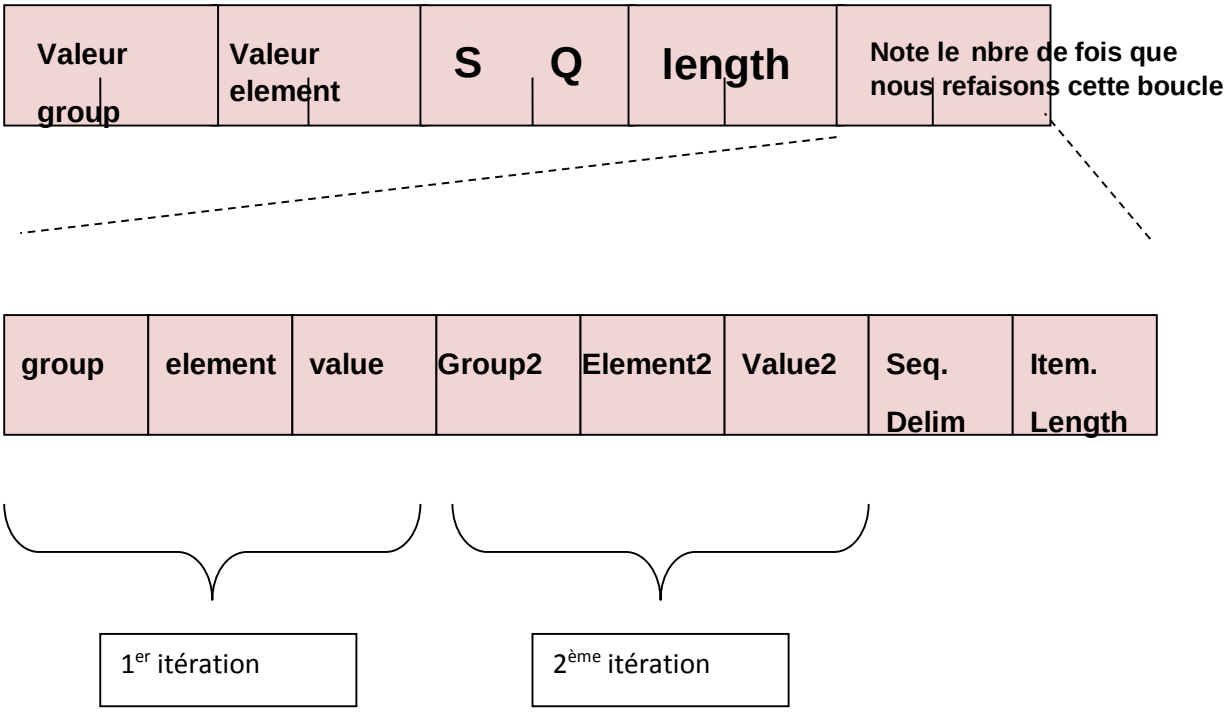
Il existe un cas spéciale dans l'encodage de DICOM quand nous avons l'attribut VR = SQ signifie Sequence of Element. Ce VR représente « Valeur de séquence de zéro ou plusieurs séquences ».

(a) Représente la façon d'être ranger dans les métadonnées :

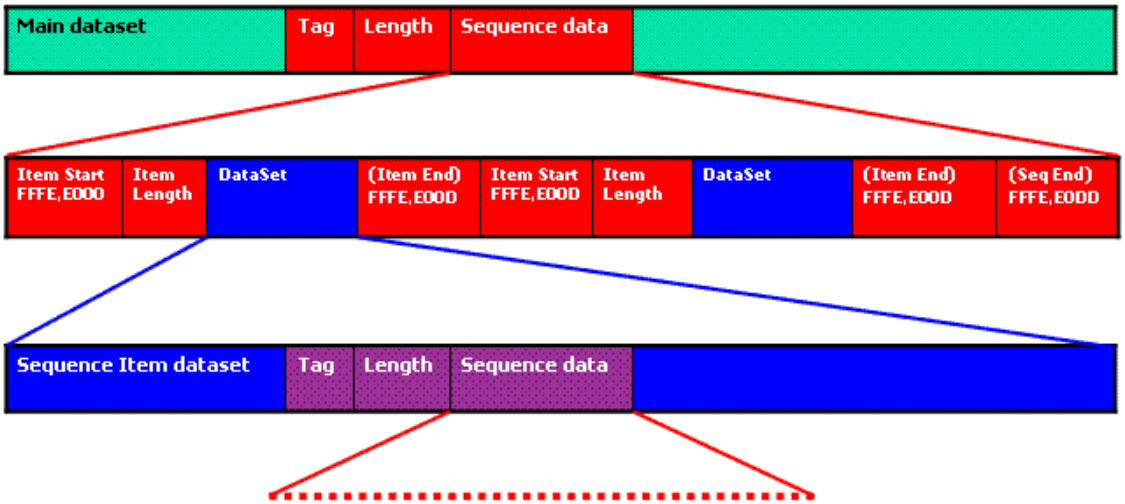
On reprend notre schéma que nous avons vu sur l'encodage d'une seule métadonnée.



Nous prenons ce cas spécifique que VR = SQ :



(b) Représentation général de ce phénomène :



On a parlé jusqu'à maintenant de l'encodage mais nous avons dit que les métadonnées sont rangées comme une file d'attente. Il a dû faire un dictionnaire des différents TAG classée selon un certain autre pour permettre d'avoir la bonne grammaire (permettre au programme de décodée la bonne valeur lu en binaire).

Tout fichier DICOM doit avoir cette structure générale pour l'ensemble des métadonnées suivante :

Ce standard est un modèle très structurés car nous avons toujours cet ordre : Patient, Etude, Série, Image.

Nous avons des UID (Unique IDentifier) pour l'étude, une série, une image.

Les UID permettent dire que l'image est unique par toutes images faites.

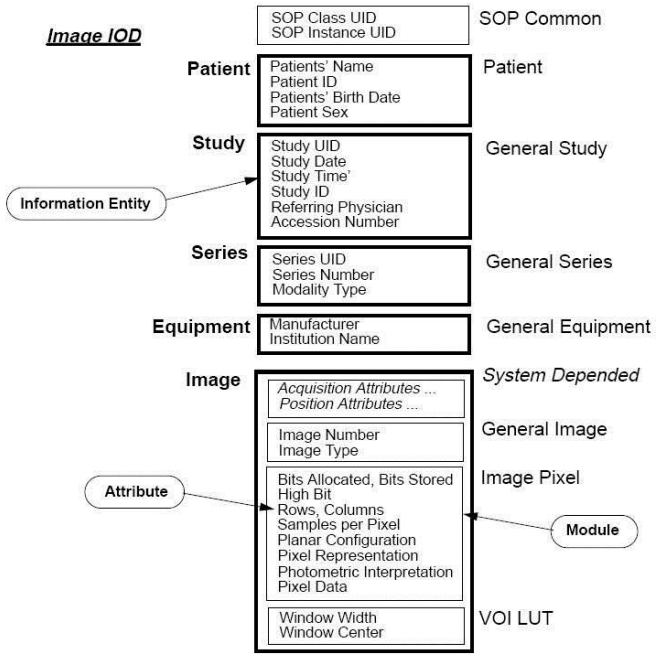


Figure 4 : schéma du classement des métadonnées

Le SOP Class sont des données qui renseignent sur la communication machine et serveur de stockage. Ils sont donnés automatiquement par la machine d'imagerie.

On a vu un petit rappel de la structure des tags. Nous aurons toujours une image DICOM sous la forme suivante :

Element Tag and Value		Binary Coding
0008,0000	**726**	**08 00 00 04 00 00 00 D6 02 00 00 ($726_2 = 2D4_{16}$)**
0008,0005	**ISO.IR 100**	**08 00 05 00 0A 00 00 00 49 53 4F 5F 49 52 20 31 30 30**
0008,0016	**1.2.840.10008.5.1.4.1.1.2**	**08 00 16 00 1A 00 00 00 31 2E 32 2E 38 34 30 2E 31 30 30 30 38 2E 35 2E 31 2E 34 2E 31 2E 31 2E 32 00**
0008,0060	**CT**	**08 00 60 00 02 00 00 00 43 54**

Explication de cet exemple :

« 0008,0000 » dans le « Element Tag and Value » column est le tag pour 0008 Group « 726 » est la valeur pour le Group length et signifie qu'il y a 726 octets dans son Group. La correspondance code binaire de son tag et la valeur sont codés en lignes « Binary coding ». Les prochaines lignes sont les tags et values peut correspondre « Specific Character Set », « SOP Class UID », « Modality », et « Study Description »

Avec dcm4che3 :

On utilise la classe org.dcm4che3.io.DicomInputStream

Et la méthode : readFileMetaInformation() qui permet de lire les métadonnées 0000,xxxx à 0008,xxxx qui sont les UID de l'images DICOM.

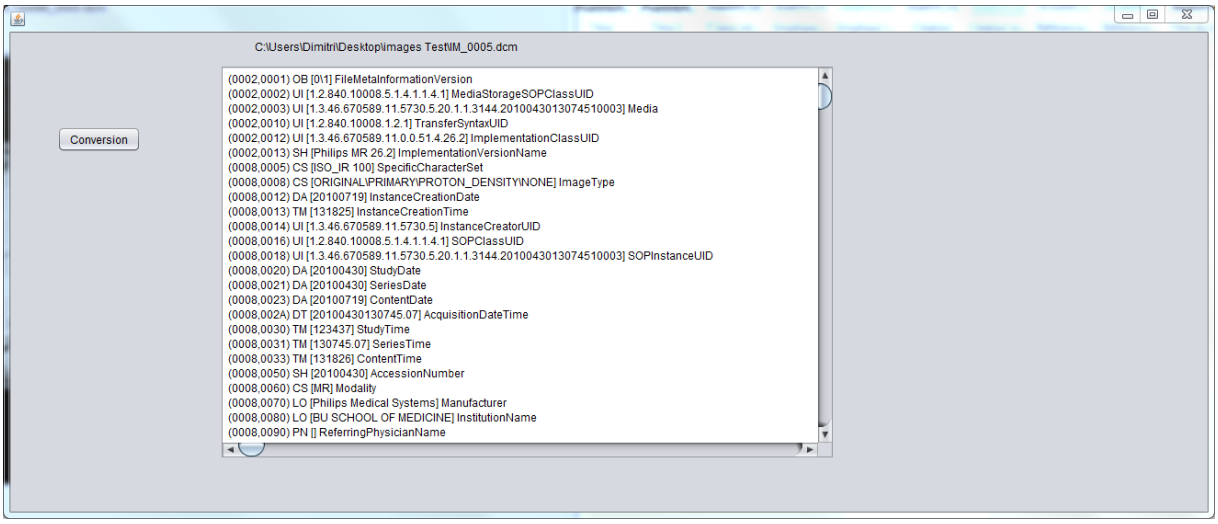
readDataset(-1, -1) qui permet de lire tous les métadonnées hors les métadinformations.

Cet exemple permet de voir comment afficher l'ensemble des métadonnées DICOM dans un String.

```
private static DicomInputStream din = null;
public static String chooserTagDicom(File file ) throws IOException{

                din = new DicomInputStream(file);
               Attributes  object =    din.readFileMetaInformation() ;
              String value = object.toString();

             Attributes object2 = din.readDataset(-1, -1);

          String value2 =
object2.toString(Tag.FileMetaInformationVersion,Tag.AcquisitionContextDescription);

        return value + value2;

         }
```

Pour faire une jolie IHM de métadonnée :

Résultat :



Le bouton conversion permet de transformer les métadonnées dans un fichier txt.

La syntaxe dans le JFrame de l'IHM de métadonnée est la suivante :

```
LireMetadataDicom medataDicom = new LireMetadataDicom();

                jLabel1.setText(input.toString());// permet d'afficher le nom du chemin de l'image

          String info2 = medataDicom.LireMetadataDicom(input);

jTextPane1.setText(info2);
```

Le fichier .txt a pour code suivant : La méthode DcmTxt permet de convertir un fichier DICOM en un texte (donc les métadonnées).

```
DcmTxt(File file, File ofile){
        try {
           String io = chooserTagDicom(file);
           ArrayList persons = new ArrayList();
 StringBuffer sb1 = new StringBuffer();
 StringBuffer sb11 = new StringBuffer();
 StringBuffer sb12 = new StringBuffer();
 StringBuffer sb13 = new StringBuffer();
 StringBuffer sb14 = new StringBuffer();
 StringBuffer sb15 = new StringBuffer();
 StringBuffer sb16 = new StringBuffer();
  StringBuffer  sb =  sb1.append("INFORMATIONS");
 StringBuffer  sb2 =  sb11.append("");
 StringBuffer  sb3 =  sb12.append("Date").append(new Date());
 StringBuffer  sb4 =  sb13.append("noun File:").append(file);
 StringBuffer  sb5 =  sb14.append("");
 StringBuffer  sb6 =  sb15.append("METADATA of DICOM 3.0:");
 StringBuffer  sb7 =  sb16.append("");
 persons.add(0,sb); persons.add(1,sb2); persons.add(2,sb3); persons.add(3,sb4); persons.add(4,sb5);
persons.add(5,sb6); persons.add(6,sb7);
  StringBuffer sbt = new StringBuffer();
iter = persons.iterator();
System.out.println(io);
while (iter.hasNext()){
   it = iter.next();
        sbt.append(it);
         sbt.append(System.getProperty("line.separator"));
   }
String textData = sbt.toString();
String text = io;
BufferedWriter out  = new BufferedWriter(new FileWriter(ofile));
out.write(textData);out.write(text);out.close(); } catch (IOException ex) {
Logger.getLogger(DcmTxt.class.getName()).log(Level.SEVERE, null, ex);        }
```

Cette méthode a pour but de lire les métadonnées dans un fichier DICOM.

```
public static String chooserTagDicom(File file ) throws IOException{
                    din = new DicomInputStream(file);
                     object =    din.readFileMetaInformation() ;
                    String value = object.toString();
                   object2 = din.readDataset(-1, -1);
                  String value2 =  object2.toString(Tag.FileMetaInformationVersion,Tag.AcquisitionContextDescription);
                    return value + value2;
                     }
```

III.2) Afficher une image dans BufferedImage:

Dcm4che3 gère toute seule avec la librairie imageio les formats d'image JPEG 2000, JPEG-loss ….

De plus, l'overlay dans les images les fichiers DICOM est géré automatiquement par ce framework.

```
    public BufferedImage chargeImageDicomBufferise(File file, int value) throws IOException  {
                                    Iterator<ImageReader> iter =
ImageIO.getImageReadersByFormatName("DICOM");//spécifie l'image
                                   ImageReader readers = iter.next();//on se déplace dans l'image dicom
                              DicomImageReadParam   param1 =  (DicomImageReadParam)
readers.getDefaultReadParam();//return DicomImageReadParam
                                //      Adjust the values of Rows and Columns in it and add a Pixel Data attribute with the byte
array from the DataBuffer of the scaled Raster
                                    ImageInputStream iis = ImageIO.createImageInputStream(file);
                                readers.setInput(iis, false);//sets the input source to use the given ImageInputSteam or other
Object
                                    BufferedImage image = readers.read(value,param1);//BufferedImage image =
reader.read(frameNumber, param); frameNumber = int qui est l'imageIndex
                                    System.out.println(image);//affichage au terminal des caractères de l'image
                                    readers.dispose();//Releases all of the native sreen resources used by this Window, its
subcomponents, and all of its owned children
                                   return  image;
                               }
```

D'autres codes dans la classe DicomBuffered.java.

III.3) Manipulation métadonnées:

Regarder le fichier DisplayTag.java.