How XNAT translates DICOM metadata

The DICOM standard includes figures describing the "DICOM model of the real-world."

This is not identical to the XNAT data model, and in fact the XNAT data model originated before XNAT support existed for DICOM. (The MR scan and session types were based on Siemens Analyze, though they've become more DICOM-ish over time; PET was based on ECAT and still hasn't drifted much DICOMward.)

Apart from the differences in the data model, the concrete representation of XNAT's data model is vastly different from how DICOM is represented. DICOM metadata storage is massively denormalized, with every image file containing all of the metadata pertaining to that image; whereas the XNAT XML is a unified, compact representation of the experiment metadata. *Notes: 1. this is the last time you will hear me call XML "compact"; 2. I'm asserting that the experiment XML is the canonical representation of XNAT metadata. You could argue for other canonical representations (e.g., the SQL database) but you'd be wrong.*

The project dicom-xnat, composed of the subprojects dicom-xnat-mx, dicom-xnat-sop, and dicom-xnat-util, defines our mapping from the DICOM data model to XNAT, and from individual DICOM metadata elements to fields in an XNAT experiment document. The primary interface to dicom-xnat is through the class org.nrg.dcm.xnat.DICOMSessionBuilder, defined in dicom-xnat-mx. When the DICOM session builder is called, it walks the experiment file tree and creates a temporary HSQL database with one row per DICOM file and one column per DICOM attribute that might contribute to an XNAT field. (In some cases, this is instead one column per translated XNAT field value; this is common for HCP customizations that use fields parsed from Siemens shadow headers.) This database is then used to compute the XNAT metadata fields: by series for scan-level attributes, and by study for experiment-level attributes.

The session builder is normally run by XNAT when the C-STORE session timeout expires, or when the upload applet indicates that session transfer is complete. The session builder can also be invoked on archived sessions via the REST API, by issuing a (blank entity) PUT to the session resource with the query parameter pullDataFromHeaders=true .

Customizing metadata translation

To add a new XNAT attribute:

1. Add the attribute to the schema
    a. It's possible to use addparams instead; there's a lot of this in HCP MR scans. This is such a common thing to do that there's a type (Xnat AttrDef.AddParam) and a utility method (wrap()) to make it easy.
2. Write the attribute conversion class
    a. This is complicated. "Attribute translation" below provides some details and lots of examples.
    b. Often there's a class that already does what you want. Be sure to look through XnatAttrDef and the various attribute lists to see if what you need is already implemented.
    c. If you're doing simple text conversion, you won't even need a class, you can just provide the XNAT attribute name and the DICOM tag
3. Add the attribute to the session/scan class (see "Attribute definition" below), as one more s.add(...) line
4. Rebuild the dicom-xnat-mx jar, changing the version number if that makes sense for you
5. Replace plugin-resources/repository/dcm/jars/dicom-xnat-mx-... with your modified version
6. Update and restart the webapp

Mechanics of metadata translation

The heart of the DICOM-to-XNAT metadata translation code is the class DICOMSessionBuilder, defined in the project dicom-xnat-mx. Most of the action takes place in the method call(), which returns a modality-specific subclass of XnatImagesessiondataBean:

- First, makeSessionBean effectively looks at the set of SOP Class UIDs defined for the project, uses org.nrg.dcm.SOPModel to determine what the experiment type should be, and creates a corresponding bean.
- Once the session type is determined, we loop over its session-level attributes, calculating values for those fields and assigning them in the experiment bean. prearchivePath needs special handling, so that attribute value is returned from setValues instead of being quietly set in the bean like all other values.
- Next, we find the contained Series Instance UIDs and Series Numbers to determine which scans are to be built. This gets rather more complicated when there are multiple series (as defined by Series Instance UID) with the same Series Number.
- Once the scans are identified, we create and invoke a DICOMScanBuilder for each scan to build the scan bean.

The scan builder is broadly similar to the session builder, though there are differences in the details:

- Two scan-level attribute values (ID, parameters/addParam) need special handling and thus get returned from SessionBuilder.setValues()
- The number of frames can't be evaluated until all the catalog resources have been built, so a closure (really a Callable<Integer>) is passed to the constructor (as nFrames) and then evaluated after all the resources have been added.

The logic for mapping DICOM modalities and SOP classes to XNAT types is accessed via static methods on SOPModel, but defined in text resource files interpreted at class load time.

As a practical matter, most metadata translation customization can be done without modifying any of this base functionality. Changing the details of file storage (e.g., by replacing the current catalog file structure) will likely require changes to at least DICOMScanBuilder.

Attribute definition

The translation operations that build individual XNAT attributes from DICOM metadata are specified in modality- (MR, PET, CT) and level- (session, scan, image) specific classes in the package org.nrg.dcm.xnat of dicom-xnat-mx.

For example, all session-level types (mrSessionData, ctSessionData) build on a base of attributes defined in ImageSessionAttributes. The attributes defined in this class are a representative sample of attribute translation definitions.

- all attribute definitions get added to a class-static container s using the method add
- the simplest translations, copying the text from the DICOM field to the XNAT attribute, can be made with just the XNAT field name and the DICOM tag:
  `s.add("acquisition_site", Tag.InstitutionName);`
  - which is API shorthand for `s.add(new SingleValueTextAttr("acquisition_site"), new FixedDicomAttributeIndex (Tag.InstitutionName))`
- more complicated translations are defined in their own classes: XnatAttrDef.Date, XnatAttrDef.Time convert from the idiosyncratic DICOM date and time types to the somewhat-less-weird XML/XSD types

The more common point of customization is the modality-specific class. Let's consider MRScanAttributes:

- set of attributes starts with generic image attribute definitions in ImageScanAttributes
- most attributes are simple (coil, parameters/sequence, parameters/imageType, ...)
- some are complicated, and those translations are implemented in their own classes (VoxelResAttribute, OrientationAttribute, ...)

Attribute translation

Classes that define the translation from DICOM metadata to one XNAT attribute must implement the interface XnatAttrDef or its superclass ExtAttrDef<DicomAttributeIndex>.

- Note that DicomAttributeIndex is not a simple DICOM tag, but rather an active component that can get a particular attribute out of a DICOM object. This usually doesn't matter but is important for complicated DICOM metadata like fields in Siemens shadow headers.
- The simplest attribute definition class is XnatAttrDef.Text, which is effectively just SingleValueTextAttr<DicomAttributeIndex>. Look at the SingleValueTextAttr source (in ExtAttr) to see how simple attribute construction works.
  - A single XNAT attribute value may be built from multiple values of the underlying, "native" attribute (in this case, DICOM).
  - The method used to aggregate native values is foldl, which takes as argument the aggregated value so far (a), plus a map (m) of one group of relevant native attributes, and returns a new aggregated value that takes into account the value(s) in m. *Note to functional programming aficionados: foldl is misnamed, as this method really acts as the function argument to folds. Oops. Also, there is no defined order in which values are returned, so the operation should be at least commutative.*
  - The initial value of the aggregator a is provided by the method start.
  - SingleValueTextAttr has a null initial aggregator, and assumes that all values folded in will be identical and non-null.
  - When all native values have been folded in, apply is called to turn the aggregator into a final XNAT attribute value. Here this is (nearly) trivial because the aggregator is just the text value of the attribute.
- A slightly more complicated case is XnatAttrDef.Date, which needs to translate from DICOM date format to XSD.
  - the aggregator for Date is the DICOM date string representation; the initial value returned from start() is null
  - the folder function foldl is similar to the Text folder, in that it verifies that there is a single non-null value
  - Finally, apply converts the DICOM date string to XSD date format
- OrientationAttribute does a fairly complicated conversion from a single DICOM attribute (Image Orientation Patient)
  - DICOM field is a vector of six values (cosine vectors for patient's orientation relative to the scanner)
  - the XNAT attribute is Siemens Analyze style: Tra (transverse), Sag (sagittal), or Cor (coronal)
  - foldl does the math to convert DICOM to Analyze, while also checking for null or conflicting values
  - apply turns the Siemens-style orientation string into an ExtAttrValue
- VoxelResAttribute translates multiple DICOM fields into a single XNAT attribute
  - the aggregator is a Map that is either empty (as initialized by start()) or has keys "x" "y" and "z"
  - foldl extracts the x-,y-, and z- voxel resolutions from the DICOM fields Pixel Spacing and Slice Thickness and puts them into the aggregator map
  - apply turns the aggregator map into an ExtAttrValue that has no text value but has attributes "x" "y" and "z"

The most complicated custom attribute definitions are in the HCP IntraDB. The attribute parameters/deltaTE, derived by the class DeltaTEAttr, has a somewhat complex value aggregation.

- deltaTE is defined only when there are two distinct values of TE in a single series; in that case, the value of deltaTE is the nonnegative difference.
- the initial aggregator returned by start is a Set
- foldl adds each TE value to the Set
- apply verifies that there are exactly two values, and returns the absolute value of the difference