Doms Views and Update Tracking

Fedora Repository Views

DOMS employs an overall atomistic data model. Atomistic data models are much more flexible than traditional compound data models, but they have one big (and largely unmet) challenge. When working with data objects you will frequently need to operate on a number of objects as if they were a common whole. The easiest usecase for this is the public dissemination of data. If the data that should go into one Dissemination Information Package is distributed over several objects, the system needs to understand this. Search indexing is another usecase. Search services tend to use a flat index, each record contain all it's metadata.

The solution to this is the concept of repository views.

Theoretical basis

A repository contain data. This data can be separated into a number of records. A record does not nessesarily correspond to a data object, but is some atomic, selfcontained entry. As they are atomic, they cannot reasonably be further broken down. As they are selfcontained, they are only weakly linked to other entries. A repository view is the mapping from the repository data into these records.

What constitues atomic selfcontained entries are dependent on the reason for accessing the repository. A search engine harvester might want to see one kind of records, while an export function might want another. We call such reasons "view angles". The mapping of data into records is dependent on the view angle.

Fedora Views

Fedora is a repository not just of data, but of digital objects. So, the view mapping should be from a number of objects into a record of some format.

I assume A to be a data object. A reasonable requirement is that for an object to be in the view of A, it must be related somehow to A. Thus, A is connected through some chain of relations, to every other object in it's view.

The second requirement, and this is very fundamental, is that **A does not know it is being viewed**. A is just a data object. It cannot be expected to keep up with new ways of accessing the repository, and new ways to view the data. So, A must not store any information that pertain solely to this or any other view angle. The relations of A should only be structural, in regards to the data it contains.

So, finding the view of A seems an impossible task, but it is not. For while the second requirement forbids A from knowing about the view angle, the class of A could. In Fedora, the classes of data objects are represented by content models. So, the content model(s) of A could know about this and other view angles of A. But a content model cannot say anything about A specifically, it can only describe the entire class of objects like A. So what it can do it annotate the relations of A. It could say "For this class of objects and this view angle, these structural relations denote references to other objects that are in the view."

This naturally lends itself to a recursive approach. The view of A is A plus the view of any object related to A through such an annotated relation.

But the angle one views the repository might also affect the number of entries seen. The above, recursive approach will always lead to one entry per data object. The remedy for this is to mark some classes as Entries for a certain view angle. This means that to compute the records for a given view angle, the view of all objects of a class that is an Entry should be computed. This is the view of the repository.

Fedora Implementation

This section describes how the above could be implemented in Fedora.

Entry Declaration

It is very simple for a content model to declare itself to be an Entry for a view angle. All it has to do is have a literal relation in the RELS-EXT datastream, by the name "isEntryForViewAngle", in the view namespace, to the literal name of the view angle.

Add this relation to any content models that should describe entries for the view angle named GUI.

```
<view:isEntryForViewAngle xmlns:view="http://doms.statsbiblioteket.dk/types/view/default/0/1/#">GUI</view:
isEntryForViewAngle>
```

Annotated Relations

To annotate relations, a special datastream have been introduced, called "**VIEW**". This datastream should exist in the content models, and the name have been made Reserved.

It is basically a list of view angles, and the relations that should be view relations for each. There is a little twist, though. Above, we only defined that an object should be related through some chain of relations to every object in it's view. We did not specify that the direction of these relations. So, if we have the objects A and B, and B have a relations #relatesTo to A, B could still be in the view of A. And indeed, A does not have to be in the view of B, even if B is in the view of A.

To achieve this, the view datastream allows you to annotate incoming relations, as well as outgoing.

The schema for the VIEW datastream is as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
targetNamespace="http://doms.statsbiblioteket.dk/types/view/default/0/1/#" xmlns="http://doms.statsbiblioteket.
dk/types/view/default/0/1/#" elementFormDefault="qualified" attributeFormDefault="unqualified">
        <xsd:element name="views" type="viewsType"/>

        <xsd:complexType name="viewsType">
                <xsd:sequence>
                        <xsd:element name="viewangle" type="viewType" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="viewType">
                <xsd:sequence>
                        <xsd:element name="relations" type="relationsType" minOccurs="0" maxOccurs="1"/>
                        <xsd:element name="inverse-relations" type="inverse-relationsType" minOccurs="0"
maxOccurs="1"/>
                </xsd:sequence>
                <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:complexType>

        <xsd:complexType name="relationsType">
                <xsd:sequence>
                        <xsd:any namespace="##any" processContents="skip" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="inverse-relationsType">
                <xsd:sequence>
                        <xsd:any namespace="##any" processContents="skip" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>

</xsd:schema>
```

Example of a VIEW datastream

This is an example of how the VIEW datastream could look for the view angle GUI.

```
<view:views xmlns:view="http://doms.statsbiblioteket.dk/types/views/0/1/#">
        <view:viewangle name="GUI">
                <view:relations>
                        <doms:hasFile xmlns:doms="http://doms.statsbiblioteket.dk/relations/default/0/1/#"/>
                </view:relations>
                <view:inverse-relations>
                        <doms:isPartOfCollection xmlns:doms="http://doms.statsbiblioteket.dk/relations/default/0
/1/#"/>
                </view:inverse-relations>
        </view:viewangle>
</view:views>
```

The GUI view angle of this object encompass the object itself, and the GUI viewangle of any objects that the object has a "doms:hasFile" relation to and any object that has a "doms:isPartOfCollection" relation to this object.

Calculating the view

The procedure to calculate the total view of a object is detailed in this bit of pseudo code. It basicly performs a depthfirst search of the objects. The order of the objects in the View does not carry any sort of meaning, and will be random.

```
Set<Object> visitedObjects;

List<Object> CalculateView(Object o) {
        List<Objects> view = new List<Objects>();

        if (visitedObjects.contain(o){
                return view;
        }

        visitedObjects.add(o);
         if (o.isDeleted){
                return view;
        }
        ContentModel c = o.getContentModel();
        List<Relation> view-rels = c.getViewRelations();
        List<Relation> object-rels = o.getRelations();

        for (Relation r : object-rels){
                if (view-rels.contain(r)){
                        view.addAll(CalculateView(r.getObject());
                }
        }

        List<Relation> view-invrels = c.getInverseViewRelations();
        List<Relation> object-invrels = o.getInverseRelations();
        for (Relation r : object-invrels){
                if (view-invrels.contain(r)){
                        view.addAll(CalculateView(r.getSubject());
                }
        }

        return view;
}
```

Content Model Inheritance/Multiple Content Models and Views

DOMS employ inheritance for content models, as detailed in Doms ECM Ontology This interferes with the View system. Even without inheritance, Fedora 3 allows an object to have multiple content models. Each of these could specify the view of the object.

If a data object have multiple content models, through inheritance or any other way, the following rules should be followed to resolve ambiguity.

- When finding the annotated relations for a given view angle for a data object, get the list from each of the content models, merge it to one list and remove the duplicates. These are the view relations of this object. Do the same with the inverse relations.
- If a content model marks an object as an Entry for a given view angle, the object is an entry. It does not matter if it has other content models that does not mark it as an entry. An object can of course be an entry for several view angles.