Model Catalogue - version2

David Milward

1 Architecture

The model catalogue version 2 is based on the existing model catalogue, but we are building it from scratch using the eclipse EMF model driven stack so that individual aspects of the model can be isolated and for academic reasons studied separately. As new designed elements are proved in the model catalogue 2 stack, we plan to adapt the existing Grails toolkit accordingly, but on an incremental basis.

The main classes in the metamodel are shown, together with their inheritance hierarchy in figure 1. These are a *CatalogueCore* class which roughly corresponding with the existing abstract *CatalogueElement* class, it is intended to be an abstract class to which administrative behaviours can be added at a later stage, these will then be available in all the subclasses in the catalogue. We summarise these classes below:

- Catalogue Core - An abstract class allowing all other classes to be administered, annotated, constrained and associated.
- DataModel - the main modelling artefact, all other entities will be a part of a DataModel
- DataConcept - An abstract class which allows all its child classes to be treated as concepts.
- DataSection - A grouping class for parts of a model, it can contain other DataSections, and DataClasses.
- DataClass - A container for DataElements.
- ValueDomain - A structure for the value/ representation of the DataElement.

Figure 2

The *CatalogueCore* class is the *parent* of both the *DataModel* and *DataConcept* class, which are shown in figure 3 The *DataModel* is a container class for all the models in the model catalogue, is roughly equivalent to the current *Classification*; *DataModels* will be the *entities* that the model catalogue will manage.

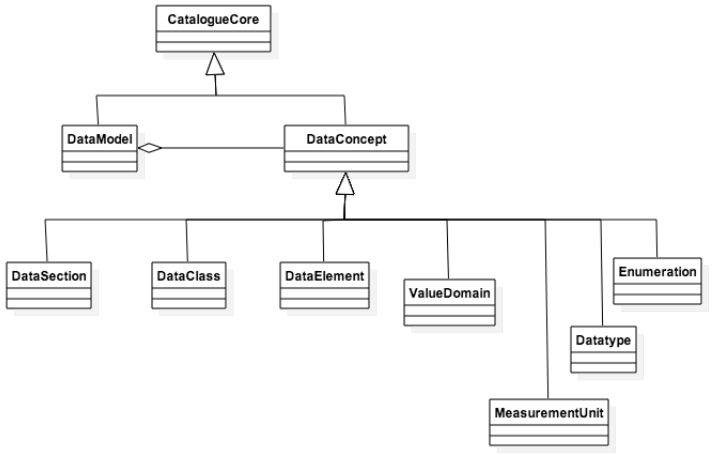

Figure 1: Core Classes - Inheritance Model

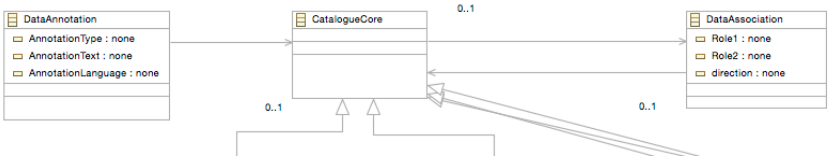

Figure 2: Core Classes 1

1.1 Groupings inside the DataModel

A *DataAnnotation* class is needed to add annotations to any of the model elements in the model catalogue, and this can include constraints. We considered adding in a spearate constraint entity, however unless we are going to constraint the constraint to a particular language it seems acceptable to adopt the UML practise of including a constraint in an annotation. The annotation will have a type of either text or constraint (not shown yet) and then it will have the annotation language, which depending on the type will either refer to a spoken language (such as German) or a constraint language (such as OCL).

The class *DataConcept* is an abstract class which is the parent of *DataSection*, *DataClass*,*DataElement*, *ValueDomain*. The main idea behind having a single abstract parent class *DataConcept* is to capture the notion that *concepts* may be represented by any one of these classes within a model.

The *DataElement*, *ValueDomain* are directly derived from the ISO11179 metadata registry standard, and correspond closely with the definitions provided within the standard. A data element in the current catalogue *has* a single value domain, this is a one-to-one containment relationship.

A *DataClass* is simply a container for *DataElements*, it's attributes are limited to *DataElements*, and cannot be *DataClasses*. There is no upperbound on

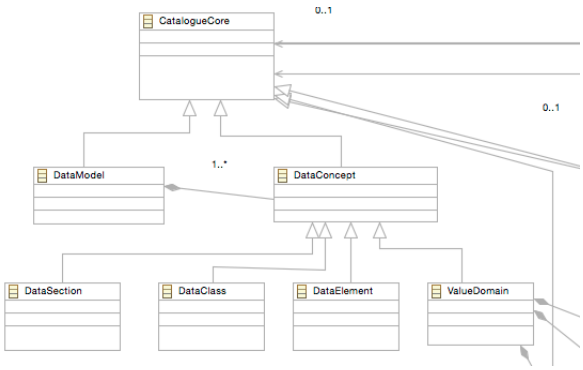

Figure 3: Core Classes 2

the number of *DataElements* that can be contained in a *DataClass*. Neither *DataClass* nor *DataElements* have an inheritance mechanism, although they can be associated using the *DataAssociation* mechanism described earlier.

The *DataSection* is a grouping element for *DataElements* and *DataClasses*, it corresponds to section in a documents and is needed as people will always section up large datasets into small digestible *chunks*, normally these will be semantically related and stretch to about 7 entities. It seems sensible therefore to define a separate *grouping* entities to enable this process.

1.2 Importing

Importing can be carried out on any entity which is a child of the CatalogueCore entity, so that while whole DataModels can be imports, so too can DataSections, DataClasses, DataElements and ValueDomains. The standard package notation will apply, so that DataModel will be named using a name and a namespace (normally a URI), and this can be extended using dot notation, so that a DataModel will be identified by **DataModelName.example.org** and a section could be **DataSectionName.DataModelName.example.org**, and so forth.

An import will be a reference to an existing finalized DataModel, so that it will not be possible to change any entities within that imported DataModel. If the imported items need to be changed then the *Cloning* operation is required.

1.3 Cloning

A clone will act in the same way as an import in terms of specifying the elements to be brought into the new DataModel, however these elements will be copied across as *new and draft* elements into the new DataModel. They can therefore be changed and altered, they may retain an association with the old DataModel, however the association will be a *basedon* type rather than *is*.

1.4 Value Domains

Value domains are entities which define how a *DataElement* is represented in the underlying system, and to do this they can contain *Datatypes*, *Enumerations* and/or *MeasurementUnits*.

In the simplest example a *DataElement* - say speed, can be represented by a *ValueDomain* which has an integer *Datatypes*. A *ValueDomain* entity being descended from a *CatalogueCore* can then have a Constraint type of *DataAnnotation* added which limits the speed to a value relative to the system under question, for boats perhaps a speed with an upper limit constraint of 600 (511 being the current world water speed record), and *MeasurementUnits* of kph.

The idea of a separate entity for measurement units is not completely neccessary as we could put that information into a *DataAnnotation*.

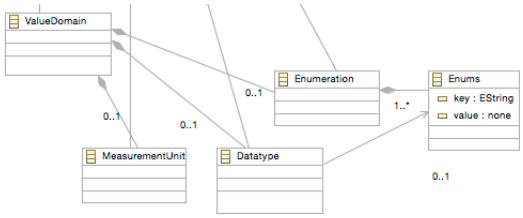

Figure 4: Core Classes 3