EDIT - bug #597

DefinedTerm problem solved

02/12/2009 12:19 PM - Andreas Müller

Status:	In Progress	**Start date:**	
Priority:	Priority13	**Due date:**	
Assignee:	Andreas Müller	**% Done:**	0%
Category:	cdmlib	**Estimated time:**	0:00 hour
Target version:	CDM UML 5.43		
Severity:	critical	**Found in Version:**	

Description

Having extendable defined terms that can be used flexible even in a none persistent environment and with a default initialization is more or less impossible.

A solution must be found using e.g. a mixed model for defined terms.

see DefinedTerms

Dear Ben, all,

Thank you very much for your long and comprehensive answer.

Yes, I think you are right that 1) and 2) are more or less contradiction or need an extreme effort to implement.
So after a long time of scratching my head and being reluctant I started to adopt the idea of enumerations for some classes.
I put up a wiki site (DefinedTerms) to collect some useful information and to continue the discussion about which class should be implemented which way.
At the moment it looks to me, that there are mainly 4 or 5 classes to be implemented as enum.
2 or 3 classes still unclear or implemented with initialization and static methods (or something similar) All the rest implemented as ordinary classes with cascading etc.

The most critical classes are Rank and NomenclaturalStatusType I think as they are not so easily made enumerations but at the same time needed for the domain logic.

I already started to implement NomenclaturalCode as an enum to gain some experience. For this I created an interface IDefinedTerm that is supposed to be implemented by all the defined term class types.
I think I will continue to implement one of the relationships to get some more experience.

Thinks we have to clarify are at least:

1) How to handle vocabularies for the enum classes
2) How to handle representations for the enum classes
3) How to access the ordinary classes via the API
4) How to make clear that also the ordinary class instances should be reused whenever possible
5) How to handle Ranks and NomenclaturalStatusTypes ..

I hope all this will help to resolve the existing problems.

Cheers,
Andreas

Hi Andreas,

I guess I wanted to de-couple term initialization from the terms themselves, as this tight coupling ties the term entities to a particular term loading implementation, and also leads to circular dependencies between spring-managed beans and hibernate entities. By allowing terms to initialize themselves, it is much harder to de-couple the term initialization from the terms themselves (and from spring, hibernate etc).

I think that there are some options here, but each solve a different problem. So I just want to take a step back and make sure that I understand what the requirements are here:

1) That term vocabularies have certain (pseudo) static members which are always available, and behave

as though they were immutable, static instances.

2) That term vocabularies are extendable at runtime, and can be re-ordered, added to, and changed

3) That these members are available (almost) as soon as an application is started

4) That if a database exists, these default terms are persisted in the database and available for use (these

terms should not need to be persisted explicitly, they should just be_there in the database).

5) That terms have multi-lingual representations

6) That the terms are easy to use (i.e. can be used without an explicit initialization step).

Its worth pointing out here that requirements (1) and (2) directly conflict with each other. Normally, it would be an either-or situation, where data types would meet requirement 1, or requirement 2, not both. I think that it is more-or-less impossible to meet both requirements in a satisfactory way. Add Spring and hibernate to the mix, and you've got a very complex situation.

Anyway, following on from this, I think that we have some potential solutions ranging from a proximate fix which might help some tests pass, to a more ultimate fix, which might make it easier to work with the defined terms more generally.

1) In terms of fixing your current applications, you can apply your fix, but I am concerned that this will remove the ability to specify how terms are loaded. Currently, you have three options:

DefaultTermInitializer - initializes terms from files. That's it.

PersistentTermIntitializer - initializes terms from files, and persists them into the database if they are not there already
TestingTermInitializer - puts a set of terms into the database, and loads the terms from that set.

In addition, if you allow terms to auto-initialize, then I'm sure you will run into problems of terms being initialized by hibernate as part of the construction of the SessionFactory prior to other beans and services (such as the transaction manager) being created, problems related to injecting spring components into data entities, circular dependencies between spring beans and data entities, problems with controlling the order in which terms are initialized etc. Also I think that because the methods must be static, one of the implementations must be chosen over the others, which makes it hard / impossible to substitute another term initialize.

So my preference is not to follow this route. My gut feeling is that it will lead to hard-to-solve problems throughout the CDM, especially once you try to scale the application across multiple users with multiple JVMs, or single web applications that are clustered, or whatever. Thinking laterally, the current implementation, which is based on static variables is already a "bad code smell" (http://en.wikipedia.org/wiki/Code_smell) as these static variables will be shared across multiple threads. In a multi-threaded application, what happens when one thread adds a term to an ordered vocabulary and re-orders the terms, whilst another thread uses those terms? (I'm developing a migraine just thinking about it!)

2) I think that a second solution is to re-examine the requirements for defined terms. I don't know what the answers are, but the questions are:

1) Is pseudo-static behaviour more important than runtime extensibility, or vice versa?

2) Is the relative importance of the two main requirements the same for all types of defined term?

My feeling is that the answer to question 2 is no. You can see this from the way in which different term classes are used. For example, NomenclaturalCode is never going to be extended during the lifetime of the CDM (unless java still exists at the point where we discover extra-terrestrials ;-), Language is not going to be extended, it seems to me that Rank is not going to be extended very often if at all, and a lot of logic is based upon it.

Feature, on the other hand, is likely to be extended, as is State, Scope, Modifier, MarkerType. NamedArea is another likely candidate, where you might have several vocabularies.

So my suggestion is that: If it is more important that a class is extensible, then we should use java Classes, and instances (i.e. the current DefinedTermBase implementation), and avoid using static member fields. This will impose restrictions on the way which those vocabularies are used, but given that they are mutable, it is probably a bad idea to base logic in the CDM on the existence and state of specific instances of those changeable terms as we can't really guarantee what the vocabulary is going to be like at runtime. i.e. At runtime there might be 2 named area vocabularies, or six or one. It is probably a bad idea to have behaviour related a specific named area vocabulary, like "English Counties" or "TDWG areas" in a generic NamedAreaService, given that sometimes, the named areas won't belong to that specific vocabulary, or might have extra members, or members which have been changed.

If it is more important that the class is static, then we should use Java enumerations, and benefit from immutability, global static members across JVMs, no need to persist (and therefore no need to do something clever when starting up the application), the ability to query by value without needing a join (i.e. such terms would be attributes, not foreign keys in related entities). This would also provide a level of stability for some of the parser / cache strategy implementations, I think, as the set of NomenclaturalCode, Rank, etc would be restricted, so application developers can't supply new ranks which are unknown to the parser / strategy generator. It would also mean that these enums could be used throughout the CDM without worrying about transactions / lazy initialization / spring etc. etc. It would also be thread-safe.

Anyway, I appreciate that this isn't really a proper answer to your question, but I'm worried that making small changes to the DefinedTermBase / term initialization design is unlikely to address the fundamental problem, which is that the design is driven by two requirements which conflict with each other. I don't think it is necessarily a failure as a software developer to pick one requirement over the other. Maybe taking some time now to think about the problem again can help avoid problems later.

Cheers,

Ben

++++++++++++++++

From: Müller, Andreas

Sent: Tuesday, February 10, 2009 4:56 PM

Subject: [dev-cdmlib] term init again

Hi Ben,

We just tested some of the new functionality you added to the Library for the term initialization and run into some kind of problem.

As you may remember you added to e.g. RankTest the following lines:

```
@BeforeClass
public static void setUp() {
      DefaultTermInitializer vocabularyStore = new DefaultTermInitializer();
      vocabularyStore.initialize();
}
```

So you explicitly force the user to initialize the terms by hand. For sure this is a good way to force the user to think about what terms he is using. On the other hand it is something I would like to avoid as it needs more explanation how to use the library. Actually the little import applications that we have in app-import did not run anymore because before we did not have any kind of initialization (e.g. the SalvadorActivator in the berlinModel package does have the following line

```
static final NomenclaturalCode nomenclaturalCode = NomenclaturalCode.ICBN();
```

that returns a null object the way it is implemented now.

So I wonder if we could change this by implementing a generic method that calls the DefaultTermInitializer if no initialization has been done yet. E.g.:

```
  protected static Map<UUID, NomenclaturalCode> termMap = null;
  protected static NomenclaturalCode getTermByUuid(UUID uuid){
       if (termMap == null){
            DefaultTermInitializer vocabularyStore = new DefaultTermInitializer();
            vocabularyStore.initialize();
       }
       return termMap.get(uuid);
   }
```

The TermMap could be filled by setDefaultTerms

```
  @Override
  protected void setDefaultTerms(TermVocabulary<NomenclaturalCode> termVocabulary) {
      termMap = new HashMap<UUID, NomenclaturalCode>();
      termMap.put(uuidIcbn, termVocabulary.findTermByUuid(NomenclaturalCode.uuidIcbn));
..
```

//also a loop over termVocabulary.getTerms is possible or even better as you do not have to copy uuids at all

This is kind of one step back in direction of the old implementation so I would like to ask you if you this is in some way against the ideas when you refactored the term initialization.

For me there are 2 cons of the implementation:

1) It needs a bit more space, as you store not only the terms but also the Map. But I think we can live with this.

2) It may cause any kind hibernate problems because of duplicate objects that I can not predict yet.

The pros are:

1) You can use the library without any active initialazation

2) You can run the Initialization for each class separately, so not all classes have to be initialized if they are not needed. This needs some further refactoring and does not allow testing for existence on application start-up.

3) It reduces the code as lines like:

```
private static NomenclaturalCode ICZN;
```

are not necessary anymore.

Also the lines in setDefaultTerms() can be deleted (see comment in the code above)

What do you think?

Cheers,

Andreas M.

Related issues:

Related to EDIT - feature request #6794: Improve term structure	**In Progress**

History

#1 - 09/03/2009 10:29 AM - Andreas Müller

- Status changed from New to In Progress

- Priority changed from Priority14 to Priority11

- Severity changed from blocker to critical

#2 - 10/05/2009 05:44 PM - Katja Luther

- Target version changed from CDM lib Release 2.0 to CDM lib Release 2.3

#3 - 07/02/2021 05:45 PM - Andreas Müller

- Tags set to terms

- Description updated

#4 - 09/17/2021 05:39 PM - Andreas Müller

- Target version changed from cdmlib - Old Next Major Release to Unassigned CDM tickets

#5 - 09/17/2021 05:40 PM - Andreas Müller

- Related to feature request #6794: Improve term structure added

#6 - 02/22/2023 10:46 AM - Andreas Müller

- Description updated

#7 - 02/22/2023 10:53 AM - Andreas Müller

- Description updated

#8 - 02/22/2023 10:54 AM - Andreas Müller

- Priority changed from Priority11 to Priority13

- Target version changed from Unassigned CDM tickets to CDM UML 5.43

#9 - 02/22/2023 10:54 AM - Andreas Müller

- Private changed from Yes to No

#10 - 02/22/2023 10:56 AM - Andreas Müller

Ben 2009-02-13:

> Yes, I think you are right that 1) and 2) are more or less
> contradiction or need an extreme effort to implement. So after a long
> time of scratching my head and being reluctant I started to adopt the idea of enumerations for some classes.

I must admit, I didn't really want to suggest changing the defined terms as so much effort has been expended implementing them. However, the more I think about it, the more uncertain I become about being able to implement them as-was, and ensure that the system was stable - the defined terms seem to be used throughout the application, so any problems with them will occur generally.

> The most critical classes are Rank and NomenclaturalStatusType I think
> as they are not so easily made enumerations but at the same time needed for the domain logic.

I understand this.

I think it is an either/or situation - either the vocabulary is an enum, or it is represented using instances. In some cases it will be clear cut that the vocabulary should be an enum or not. In some cases it will be more ambiguous, but I think we should still try to decide one way or the other and maybe accept the restrictions.

My thought was that you could use apache-like +/- voting on a number of criteria in the not-so-obvious cases, but looking at the wiki, you've already done this.

One thought is that, if there is domain logic (i.e. logic in the domain entities), based upon particular (static) instances of a term, or logic in parsers or formatters / cache strategy generators which are set by default in domain entities, this is a fairly strong argument for using enums as it solves two types of problems (1) coupling between domain entities and the persistence infrastructure and all the issues associated with it, and (2) Problems with understanding the behaviour of that parsing (or whatever) logic. If someone has written a parser for names, then it is much easier to write and test it if you know what values in the term enumeration are and don't have to accommodate cases where an application developer or user supplies an unknown term i.e. not Rank.UNKNOWN, but an instance of Rank which the user has added.

So I guess that, even though Rank and NomenclaturalStatusType are expected to change a bit, I think it might be possible, or even benificial to make them enums (and benefit from the stability and ease-of-use), and put in place a process (perhaps trac, in the first case), to handle the desire to add new Ranks or NomenclaturalStatusTypes. It might be straightforward to say: "If you find a rank which isn't in this vocabulary, please let us know and we'll add it". This would allow the cdm developers to verify the logic based upon those enumerations e.g. check that parsing and formatting names works as expected etc. I've found that the CATE users wanted 2 ranks added where they didn't exist, but didn't care so much about the existence of ranks which they didn't use.

One thing which might help is using EnumSet's (http://java.sun.com/j2se/1.5.0/docs/api/java/util/EnumSet.html) to hold specific sub-vocabularies for things like Rank. I was imagining that you could have enumsets for Family-Group ranks, Genus-Group ranks, and Species-Group ranks (I think Family-Group etc occurs in the Zoological Code, not the Botanical Code). Equally, you might have Botanical Ranks (e.g. ranks which only occur in Botany, like lusus) or Zoological Ranks (e.g. ranks like abberation), or Cultvated Plant Ranks (e.g. grex).

> 2) How to handle representations for the enum classes

I don't think Enums can hold references to entities, so having enums returning Representations, or Media won't work. Enums in hibernate are persisted as integers fields by default, but you can set them to be @EnumType(EnumType.STRING) which persists them as a string e.g. ("IS_SAME_AS", "IS_NOT_THE_SAME_AS"), which is a bit more readable. Likewise, jaxb has an @XmlEnum enumeration.

An alternative to using Media and Representation entities to hold representations of the term which is an enum is to use ResourceBundles, which could return localized string representations of terms. This would mean adapting IDefinedTerm to look a bit like this

```
public interface IDefinedTerm {
// ... Other stuff
public String getAbbreviatedLabel(Language language);
public String getLabel(Language language);
public Set getMedia(); // or something like this }
```

Where the DefinedTerm class could delegate to the Set representations, and the enums could use ResourceBundle to get localized strings. I'm not sure what to do about images, but I think you could do a similar sort of thing, holding uris for images.

> 3) How to access the ordinary classes via the API

Perhaps the converse of of the argument that "If there is logic in the domain model based upon a particular term, then there are real benefits of implementing those terms as an enum" is "If a term vocabulary is made of instances, it is better to avoid making pseudo-static references to those terms in domain model entities". What I mean by this is that: If we implement a term vocabulary as classes, then I would try to remove logic based upon specific terms or vocabularies from the domain model classes. When I did this in CATE, I moved the logic up into the controller and view layers

with the reasoning being: These vocabularies are specific term vocabularies, used by a specific application which has specific business rules and requirements. So I ended up implementing these "vocabularies" as services - you can see an example here: http://forge.nesc.ac.uk/cgi-bin/cvsweb.cgi/cate-service/src/main/java/org/cateproject/service/enumeration/NamedAreas.java, with it's implementation h!
ere: http://forge.nesc.ac.uk/cgi-bin/cvsweb.cgi/cate-service/src/main/java/org/cateproject/service/enumeration/impl/NamedAreaLevelsFactory.java

In this scenario, controlled terms in the cdm could be split into:

Terms which the CDM depends upon to work properly, implemented as enumerations

Terms which the CDM knows about in general, but does not depend upon specific instances, or specific vocabularies to work. To make things easier, the CDM could supply "Default" vocabularies for these types of term, e.g. it makes sense that the CDM might initialize and use the vocabulary of TDWG Named Areas by default, but none of the core logic depends upon that specific vocabulary's existence - users could substitute another area scheme (and indeed, if they work on marine organisms, they probably would not want to use the TDWG codes!), or add another area vocabulary alongside it.

> I hope all this will help to resolve the existing problems.

I hope so too! I'm currently on track to finish the xml import stuff for CATE by the end of next week, so I'll have a bit of time then to help out if you need something done.

I'm happy to add some comments to the wiki if you think it is appropriate.