Design Document

for the

Clinical and Translational Sciences Data Standards Repository (ctsDSR)

Version 1.0

Table of Contents

1. Introduction 1

2. Overview of the system 1

3. System Architecture 1

I. ctsDSR Model 2

A. Administration and Identification classes 3

B. Classification classes 5

C. Naming and Definition Classes 6

D. Metadata classes 7

II. Design for the ctsDSR Model Loader 15

A. Loading data to ctsDSR 16

B. Design Goals 16

C. Architecture Overview 16

D. Design Patterns 17

E. Main Classes 18

F. UML Diagrams 19

III. Configuration Properties 21

4. Glossary 22

5. References 23

6. Appendix 24

I. XMI to ctsDSR Objects Mapping 24

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Date | Reason For Changes | Version |
| Preeti Lodha | 20/1/09 | Draft | 1.0 |
| Preeti Lodha | 22/1/09 | Changes as per comments from Srikanth Adiga | 1.1 |
| Preeti Lodha | 27/1/09 | Changes as per comments from Srikanth and Mukesh | 1.2 |
| Preeti Lodha | 28/1/09 | Changes as per comments from Srikanth | 1.3 |
| Preeti Lodha | 29/1/09 | Updation of design details in section II and III | 1.4 |

# Introduction

This document describes the design of the first prototype version of federated metadata repository, Clinical and Translational Metadata Repository (ctsDSR). The document gives an overview of the system and explains the ctsDSR architecture followed by the ctsDSR’s UML model details. The next section describes the design for the ctsDSR Model Loader which is the interface to import data to ctsDSR. The last section describes the way some of the administration data not available in the XMI is imported to ctsDSR.

# Overview of the system

The main aim of the system is to build a metadata repository which can be used to store, share, and aid reuse of metadata defined using Dynamic Extensions (DE) across applications locally and remotely in a caBIG Compatible manner. Initially the focus will be on developing a **Proof of Concept** (POC) which accepts the UML model as an XMI and loads the metadata to ctsDSR. It has been initially assumed that the new models will be independently defined in DE, exported as XMI (version 1.3) and semantically annotated using the SIW. The fully annotated XMI will then be exported from the SIW and be loaded into the ctsDSR so that metadata of the new DE model may be shared in an ISO-11179 compliant fashion. No direct communication between the DE and the ctsDSR application is needed in this POC. Validations to check if the XMI is completely annotated will be added in future versions of ctsDSR. In future the system also aims to have plugins to interact with DE and UML Modeling tools like EA and/or Argo UML.

# System Architecture

The design for the ctsDSR is based on the “caCORE” applications development framework. The model for ctsDSR is an ISO 11179 compliant model prepared based on the gap analysis done for models of existing metadata repositories like cgMDR, caDSR and DE. This model has also been reviewed by Denise Warzel who is the Project manager for Core Infrastructure, caCORE Product Line and caDSR at the NCI. Kindly refer the ctsDSR model from link available in Section 5.

ctsDSR provides an interface through which the UML models can be uploaded to the system in the form of an XMI which has been exported from DE and annotated using SIW. Important administration data which is required as per ISO 11179 and not available in the XMI is imported using a configuration file ctsDSRConfigurations.xml. Kindly refer section [3-III] for more details.

ctsDSR exposes the caGrid API (caGrid 1.2) and the caCORE API (caCORE 4.1) enabling syntactic and semantic interoperability between diverse applications. Kindly refer links for caCORE and caGrid framework available in Section 5.

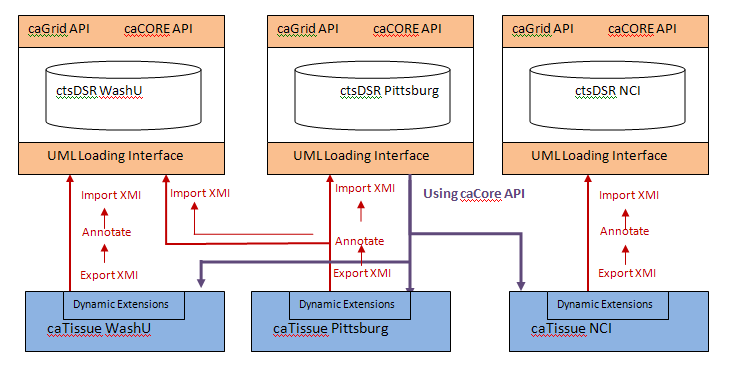


Figure ctsDSR Architecture

The diagram above explains the architecture for ctsDSR.

As depicted in the above figure, several caTissue instances deployed at different locations like WashU, Pittsburgh, NCI, etc can independently create extensions to the static model using DE. To allow other applications with similar interests to reuse these CDE’s/models defined using DE, they need to be registered in ctsDSR. This can be done by exporting the model from DE, annotating them using SIW and importing the annotated model in ctsDSR. There may be several instances of the ctsDSR deployed. The same model may be registered in one or more ctsDSRs. ctsDSR exposes these models via the caCORE and caGrid API allowing sharing of these models/CDEs without the need of registering them on caDSR.

1. ctsDSR Model

This section details out the classes in the ctsDSR model which are used to store the metadata and various other details as per ISO 11179 specification. This is a hybrid model developed based on the gap analysis done between models of already existing metadata registries like caDSR, cgMDR and DE. A detailed study of each of the models was done to find out what is mandatory as per ISO specifications and minimally required to store all details about any UML model. Based on this study and the gap analysis between the three models, we have come up with this model for ctsDSR which closely resembles the cgMDR model, with a few extensions taken from the caDSR model as per discussions with Denise Warzel, who is the project manager for caDSR at NCI.

This section uses a metamodel to describe the structure of a Metadata Registry. The registry in turn will be used to describe and model other data, for example about enterprise, public administration or business applications. The registry metamodel is specified as a conceptual data model, i.e. one that describes how relevant information is structured in the natural world. This section is sub-divided into the following

1. Administration and Identification classes: Support the administrative aspects of Items in a registry
2. Classification classes: Provide logical grouping of items
3. Naming and Definition Classes: These are used to manage the names and definitions of items in various contexts.
4. Metadata classes: Responsible for actual storage of Object Classes, their attributes, value domain, Data Element Concepts, data elements, etc.
5. Administration and Identification classes

The Administration and Identification region supports the administrative aspects of items in a registry. This region addresses:

1. The identification and registration of items submitted to the registry
2. The organizations that have submitted items to the registry, and/or that are responsible for items within the registry, including Registration Authorities
3. Contact information for organizations
4. Supporting documentation
5. Relationships among administered items.

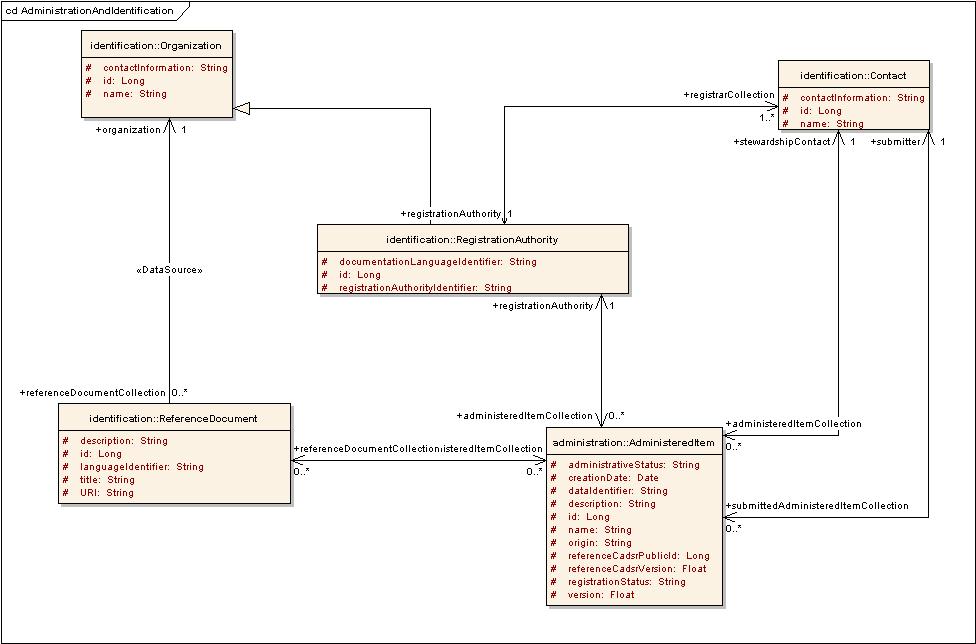


Figure : Administration and Identification

Below are the details for each of the above classes.

1. AdministeredItem: *AdministeredItem* is any **registry item** for which administrative information is recorded. All the administration data for the item is maintained in this data structure. It stores important information about a registry item like its administrative status[[1]](#footnote-2), registration status[[2]](#footnote-3), name, unique identifier, version etc. An *AdministeredItem* is *registered by* a *RegistrationAuthority* represented by the relationship *registrationAuthority*. The contact information for the person who submitted the *AdministeredItem* is represented by the relationship *submitter*. An *AdministeredItem* is *administered by* a contact represented by the relationship *stewardshipContact*. An *Administered Item* may be *described by* zero or more *Reference Documents* as represented by the relationship *referenceDocumentCollection*. It also has field’s *referenceCadsrPublicId* and *referenceCadsrVersion* which will be used to store references to caDSR public Id and Version if any are available from the XMI.
2. RegistrationAuthority: A *RegistrationAuthority* is any *Organization* authorized to register metadata. A *Registration Authority* is a subtype of *Organization* and inherits all of its attributes and relationships. An *AdministeredItem* has a *RegistrationAuthority* that is its owner, shown by the relationship *registrationAuthority*. A *RegistrationAuthority* may register many *Administered Items*
3. Contact: *Contact* is used to specify the contact information for *registrarCollection*, *stewardshipContact* and *submitter.* Contains details like the *contactName and contactInformation* which is the communication information for the person to contact*.* Every contact is a part of some *RegistrationAuthority*.
4. Organization: This contains the details of the O*rganization* which submitted the item in the metadata repository. Contains details like *name* and *contactInformation* for the *Organization*.
5. ReferenceDocument: This stores details about a document that provides pertinent details for consultation about a subject. An *AdministeredItem* may be described by one or more *Reference* *documents* which are submitted by the *Organization* submitting the item.
6. Classification classes

The Classification region provides a facility to register and administer Classification Schemes and their constituent Classification Scheme Items. Optionally, a Classification Scheme may be used to classify items within the registry.

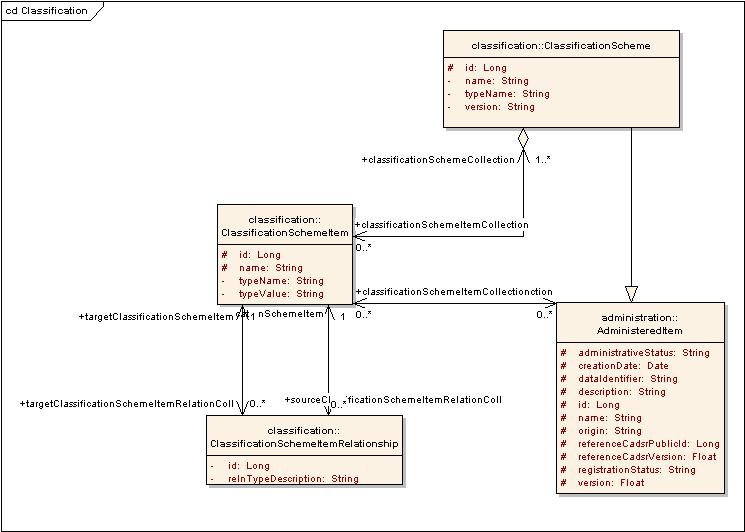


Figure Classification Region

Below is a brief description of each of the classes

1. ClassificationScheme: *ClassificationScheme* is the descriptive information for an arrangement or division of objects into groups based on characteristics, which the objects have in common. A *ClassificationScheme* may be taxonomy, a network, ontology, or any other terminological system. A *ClassificationScheme* is a sub-type of *AdministeredItem*, inheriting its attributes and relationships, which allows it to be identified, named, defined and optionally classified
2. ClassificationSchemeItem: A *ClassificationSchemeItem* represents an individual item within a *ClassificationScheme*. This may be a node in a taxonomy or ontology, a term in a thesaurus, etc. Usually stores details for packages in UML Model. An *AdministeredItem* may be classified in zero or more *Classification Schemes*, by associating it with one or more *ClassificationSchemeItem*
3. ClassificationSchemeItemRelationship: The relationship between various *ClassificationSchemeItems* within a *ClassificationScheme* is described by the *ClassificationSchemeItemRelationship*. Such relationships serve to assist navigation through a large number of *ClassificationScheme Items*. For e.g. one *ClassificationSchemeItem* may be a specialization of the other. This “IS-A” relationship is stored in the *ClassificationSchemeItemRelationship*.
4. Naming and Definition Classes

The Naming and Definition region is used to manage the names and definitions of *AdministeredItem* and the contexts for the names. It is recognized that an *AdministeredItem* may have many names that will vary depending on discipline, locality, technology, etc. In the POC we will not be concentrating much on populating these classes and maintaining alternate names and definitions. We can take this up in later versions.

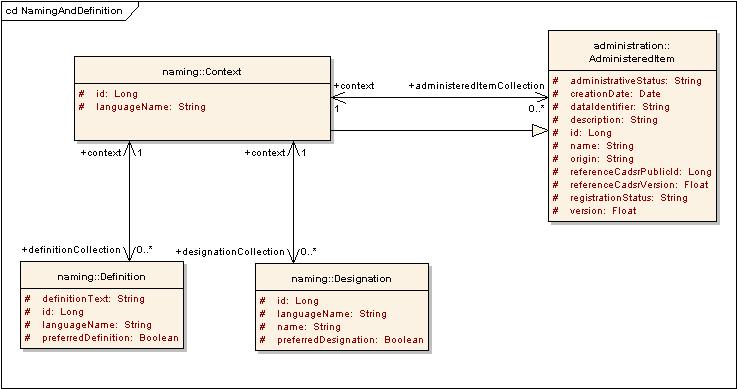


Figure Naming and Definition

Below is a brief description of the classes above:

1. Context: Each *AdministeredItem* is named and defined within one or more *Context*s. A *Context* defines the scope within which the subject data has meaning. A *Context* may be a business domain, an information subject area, an information system, a database, file, data model, standard document, or any other environment determined by the owner of the registry. Each *Context* is itself managed as an *Administered Item* within the registry and is given a D*efinition* and a *Designation*.
2. Definition: It is where the definition for an *AdministeredItem* is specified in a particular language for a particular *Context*. Where multiple *Definitions* are provided within the same context, one of them may be specified as the *preferredDefinition*
3. Designation*:* It is where the **name** for an *Administered Item* is specified in a particular language for a particular *Context.* Where multiple *Designations* are provided within the same *Context*, one of them may be specified as the *preferredDesignation*
4. Metadata classes

This section provides the basic conceptual model, including the basic attributes and relationships, for a metadata registry.

Data Element Concept

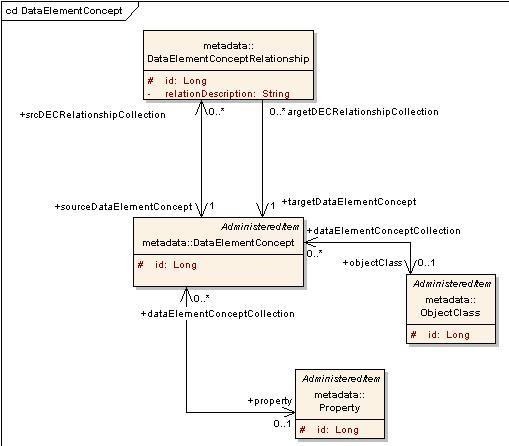


Figure : Data Element Concept

Below is a brief description of the classes above.

1. ObjectClass: This class represents and stores data about the “class” within a UML Model. Being a subclass of *AdministeredItem*, an *ObjectClass* inherits its attributes and relationships and allows it to be identified, named, defined and optionally classified within a *ClassificationScheme*. An *ObjectClass* may be registered as an *AdministeredItem* without necessarily being associated with a *DataElementConcept* or a *Property.*
2. Property: This class represents and stores data about an “attribute” of a class within a UML Model. A *Property* is a characteristic of an *ObjectClass*. It may be any feature that humans naturally use to distinguish one individual object from another. As it is a subclass of an *AdministeredItem*, a *Property* carries its own administration information, allowing it to be identified, named, defined and optionally classified within a *ClassificationScheme*. A *Property* may be registered as an *AdministeredItem* without necessarily being associated with a *DataElementConcept* or an *ObjectClass*.
3. DataElementConcept: A *DataElementConcept* may have zero or one *ObjectClass* and zero or one *Property*. The union of a *Property* and an *ObjectClass* provides significance beyond either that of the *Property* or the *ObjectClass*. Being a subclass of an *AdministeredItem*, a *DataElementConcept* carries its own administration information, allowing it to be identified, named, defined and optionally classified within a *ClassificationScheme*
4. DataElementConceptRelationship: A *DataElementConcept* may be associated with other *DataElementConcepts*, via the *DataElementConceptRelationship*. The nature of the relationship is described using the *relationDescription*.

Conceptual Domain

A conceptual domain is a set of value meanings. Many value domains may be in the extension of the same conceptual domain, but a value domain is associated with one conceptual domain. Every *DataElementConcept* is represented by a conceptual domain

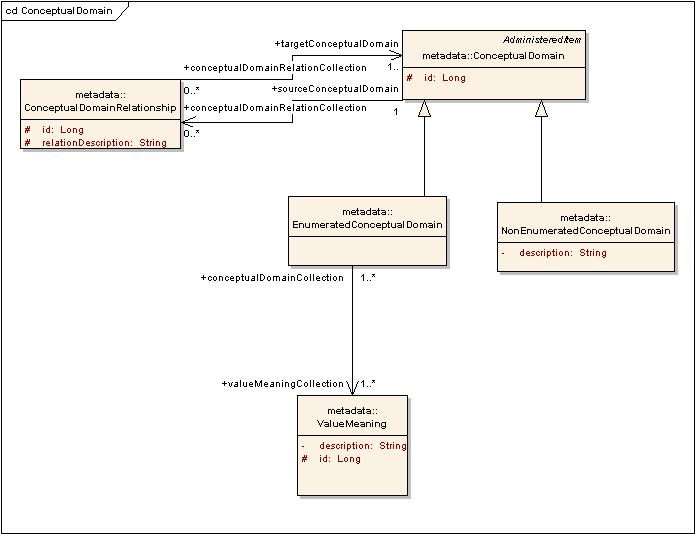


Figure Conceptual Domain

Below is a brief description of the classes above

1. ValueMeaning: This defines the meaning or semantic content of a *Value***.** It stores the set of logical values for a *Conceptual Domain*
2. ConceptualDomain: A *ConceptualDomain* is a set of value meanings. Many value domains may be in the extension of the same *ConceptualDomain*, but a *ValueDomain* is associated with one *ConceptualDomain*. As an *AdministeredItem*, a *ConceptualDomain* carries its own administrationinformation, allowing it to be identified, named, defined and optionally classified within a *Classification Scheme*
3. EnumeratedConceptualDomain: A *ConceptualDomain* sometimes contains a finite allowed inventory of notions that can be enumerated. Such a *ConceptualDomain* is referred to as an *EnumeratedConceptual Domain*
4. NonEnumeratedConceptualDomain: A *ConceptualDomain* that cannot be expressed as a finite set of *Value Meanings* is called a *NonEnumeratedConceptual Domain*. It may be expressed via a description or specification, such as a rule, a procedure, or a range (i.e., interval).
5. ConceptualDomainRelationShip: A *ConceptualDomain* may be associated with other *ConceptualDomains*, via the *ConceptualDomainRelationship* The nature of the relationship is described using the *relationdescription*.

Value Domain

One of the key components of a representation is the *ValueDomain*. A *ValueDomain* provides representation, but has no implication as to what *DataElementConcept* the values are associated or what the values mean. A *ValueDomain* is associated with a *Conceptual Domain*. A *ValueDomain* provides a representation for the *Conceptual Domain*. An example of a Conceptual Domain and a set of Value Domains is ISO 3166, Codes for the representation of names of countries. For instance, ISO 3166 describes the set of seven Value Domains: short name in English, official name in English, short name in French, official name in French, alpha-2 code,alpha-3 code and numeric code.

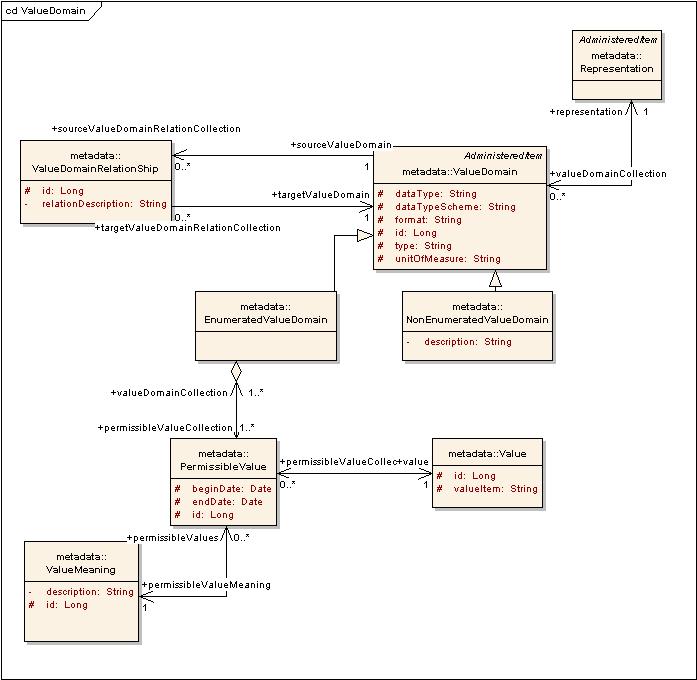


Figure Value Domain

Below is a brief description of the classes above

1. ValueDomain: It represents the values associated with an UML model attribute. It is the set of values for one or more *DataElements*. It is used for validation of data in information systems and in data exchange. It is also an integral part of the metadata needed to describe a *DataElement*. In particular, a *ValueDomain* is a guide to the content, form, and structure of the data represented by a D*ataElement*. As an *AdministeredItem*, a *ValueDomain* carries its own administrationinformation, allowing it to be identified, named, defined and optionally classified within a *Classification Scheme*
2. EnumeratedValueDomain: An *EnumeratedValueDomain* is one where the Value Domain is expressed as an explicit set of one or more Permissible Values
3. NonEnumeratedValueDomain: This is a *ValueDomain* which is expressed via a description or specification, such as a rule, a procedure, or a range (i.e., interval), rather than as an explicit set of *PermissibleValues*.
4. PermissibleValue: This class stores the links to values for a *DataElement*. A *PermissibleValue* is an expression of a *ValueMeaning* within an *EnumeratedValueDomain*. It is one of a set of such values that comprises an *EnumeratedValueDomain*. Each *PermissibleValue* is associated with a *ValueMeaning*
5. Value: This is the actual value associated with a *PermissibleValue* in an *EnumeratedValueDomain*. The same *Value* can be shared by one or more *PermissibleValues*
6. ValueDomainRelationShip: A *ValueDomain* may be associated with other *ValueDomains*, via the *ValueDomainRelationship*. The nature of the relationship is described using the *relationDescription*.
7. Representation: This is the classification scheme for representation. The set of classes make it easy to distinguish among the elements in the registry. For instance, a D*ataElement* categorized with the R*epresentation* class 'amount' is different from an element categorized as 'number'. Being a subclass of *AdministeredItem*, a *Representation* class carries its own administration information, allowing it to be identified, named, defined and optionally classified in a *ClassificationScheme*. *Representation* class is a mechanism by which the functional and/or presentational category of an item may be conveyed to a user.

Data Element

A *DataElement* is considered to be a basic unit of data of interest to an organization. It is a unit of data for which the definition, identification, representation, and permissible values are specified by means of a set of attributes. A *DataElement* is the association among a *DataElementConcept*, a *ValueDomain* and optionally a *Representation*. A *DataElement* cannot be registered as an *AdministeredItem* without being associated with a *DataElementConcept* and a *ValueDomain*.

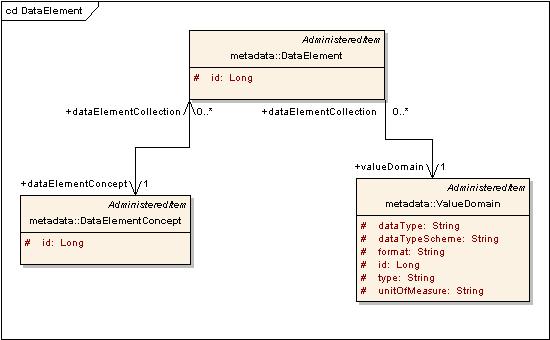


Figure Data Element

1. DataElement: . A *DataElement* is the association among a *DataElementConcept*, a *ValueDomain* and optionally a *Representation*. Being a subclass of *AdministeredItem*, a *DataElement* carries its own administration information, allowing it to be identified, named, defined and optionally classified in a *ClassificationScheme*. A *DataElement* is formed when a *DataElementConcept* is assigned a representation. One of the key components of a representation is the *ValueDomain*, i.e. restricted valid values.

Data Element Derivation

A *DataElement* may be derived from other *DataElements* by applying certain rules/formulas. It may be simple arithmetic or logical operations or some complex formulas which might be applied on one or more *DataElements* resulting in one or more *derivedDataElementCollection*

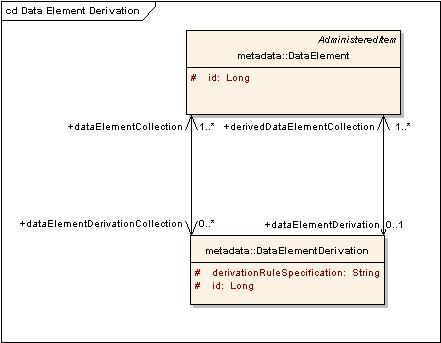


Figure Data Element Derivation

The above diagram represents the way to handle DataElementDerivations.

DataElementDerivation: It is the relationship among a *DataElement* which is derived, the rule controlling its derivation i.e. *derivationRuleSpecification*, and the *DataElement*(s) from which it is derived.

Object class Associations

This depicts the association between classes within an UML Model. One class may be related to another in one of the several ways i.e. 1:Many, Many:1, Many:Many, etc as per UML Standards.

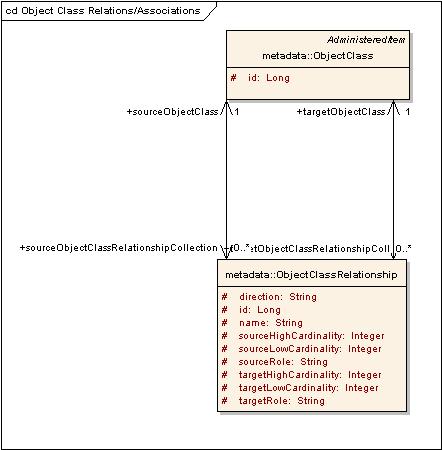


Figure Object Class Associations

1. ObjectClassRelationShip: This class stores all details about the relationship of an *ObjectClass* with another. This is not mandatory as per ISO Specifications, but has been added for integrity.

Concept Mappings

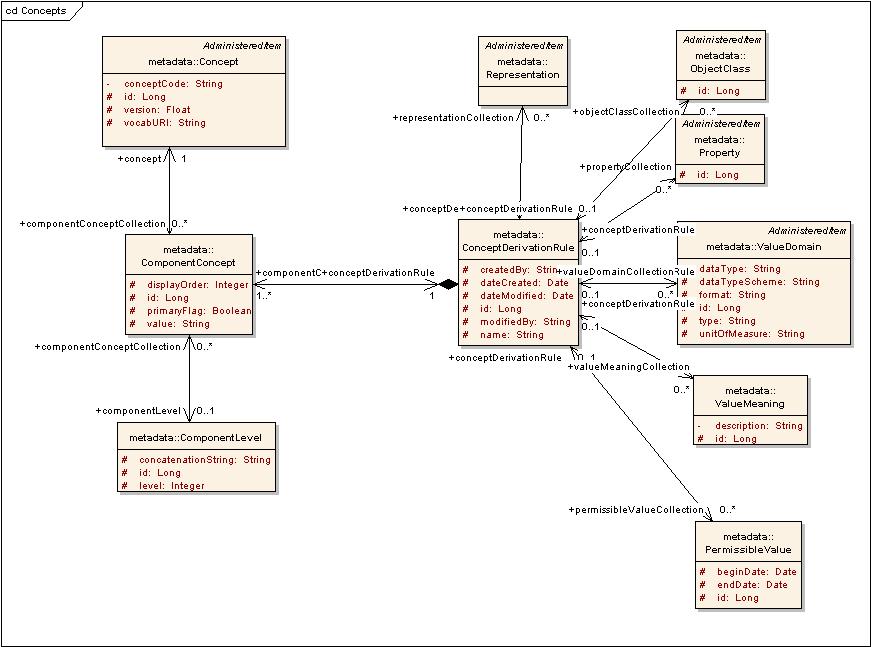


Figure Concepts

1. Concept: It is a unit of knowledge created by a unique combination of characteristics. Each *concept* is referred from a vocabulary and its combination helps in uniquely identifying any item in registry assisting its search and reuse. Each *Concept* is uniquely identified by its concept code and the *vocabURI*
2. ConceptDerivationRule: This class acts as a medium to group a set of *oncepts* associated with a registry item in a particular order.
3. ComponentConcept: This class actually stores details about the order of concepts, if it is a primary/qualifier concept, etc for a *ConceptDerivationRule*.
4. ComponentLevel: Two or more *Concepts* may be logically be grouped together within a *ConceptDerivationRule* with relations like “and”, “or”, etc. *ComponentLevel* stores details about such relationships if any exist.
5. Design for the ctsDSR Model Loader

This section elaborates on the mechanism for loading data, design considerations and classes involved in the ctsDSR Model Loader utility used to load metadata into the ctsDSR. The design for the ctsDSR Model Loader resembles the design for caDSR UML Loader. It follows the same design architecture as the caDSR UML Loader with the removal of a few components related to XMI validations, etc

1. Loading data to ctsDSR

ctsDSR Model Loaderis the tool used to register UML model metadata into the ctsDSR. ctsDSR has at its backend a MySQL DB as the data repository. ctsDSR uses the Hibernate framework to persist the objects to the backend repository.

The name of the XMI file containing the UML model, the name and version of the project being uploaded are input as command line arguments to ctsDSR Model Loader. Other administration details which are not available as a part of the XMI are accepted as a configuration XML ctsDSRConfigurations.xml available in the class-path. This file contains the configurations for administrative data e.g. organization registering the model, contact person, etc with some default values which can be edited as required. Kindly refer section 3(III) for further details on the configuration file.

The model XMI and the configurations XML are parsed by the ctsDSR Model Loaderto build ctsDSR model objects and persisted to the backend database using Hibernate.

1. Design Goals

The purpose of the architecture is to keep ctsDSR Model Loader modular. The design makes extensive use of the strategy pattern, allowing each portion to be replaced by another, if need be. From a design perspective, the ctsDSR Model Loader makes no assumption that the data will be read from XMI or persisted to a ctsDSR model.

1. Architecture Overview

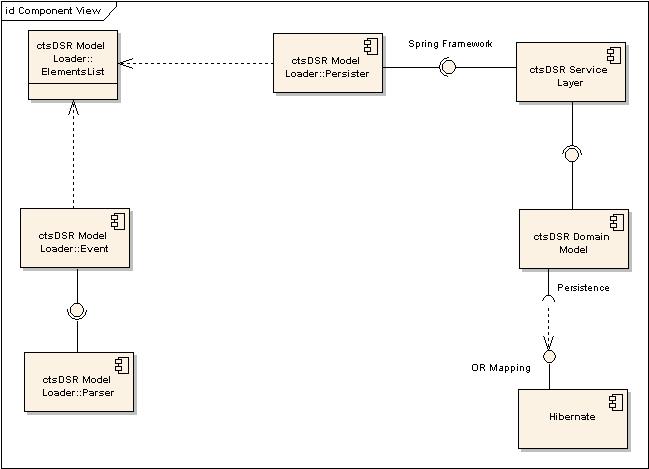


Figure ctsDSR Model LoaderArchitecture

ctsDSR Model Loader consists of four major components:

* The Parser component responsible for opening and parsing the XMI and XML files
* The Event component manages events sent by the parser component
* The ElementsList component stores in ctsDSR format all the UML/configuration elements that make up the XMI/XML file
* The Persister component responsible for sending ctsDSR objects to the service layer.

ctsDSR Model Loader uses the ctsDSR Service layer to interact with the persistence layer. The service layer uses Spring Framework to expose services and Hibernate for persistence.

1. Design Patterns

Singleton

The singleton pattern solves the issue of having multiple instances of a class within one virtual machine. ElementsList implements the singleton pattern since it is essential that ctsDSR Model Loader classes share one and only one instance of ElementsList.

Strategy

The strategy pattern is useful when one wants to replace one algorithm by another one without affecting the client using them. XMIParser, XMLConfigurationParser, UMLDefaultHandler and UMLPersister all implement the strategy pattern. Each one of these classes can be independently replaced by another implementation.

1. Main Classes

As depicted in figure 13 above, the main interfaces involved are the *Parser*, *EventHandler*, *ElementsList* and the *Persister*. The *Parser* parses the XMI and XML and sends notifications to the *EventHandler* in form of events like *newAttributeEvent*, *newAssociationEvent*, etc. The *EventHandler* accepts the events and transforms them to ctsDSR domain objects. These objects are stored in the *ElementsList* singleton object which is accessed by the *Persister* to persist objects to datasource.

1. Parser

The parser interface defines the parse(String filename) method. Classes implementing this interface define what technology to use for parsing the input file. XMIParser implements the interface and as the name suggests, expects an XMI file. It uses the Netbeans MDR API to read XMI. XMLConfigurationParser also implements the interface and parses the XML file containing configurations (ctsDSRConfigurations.xml). It uses the DOM parser implementation to parse the XML.

Parser also defines the setEventHandler method. The parser creates events and fires them to a handler. XMIParser creates UML related events, such as NewAttribute or NewAssociation. XMLConfigurationParser creates other events, such as NewRegistrationAuthority or NewSubmissionContact. Parser expects an UMLHandler generalization of EventHandler.

1. EventHandler

The Parser creates events and fires them to the EventHandler. Events are UML/configuration related, such as NewClassEvent, NewAttributeEvent, NewRegistrationAuthorityEvent,. The UMLDefaultHandler implementation of EventHandler has the task of receiving events and transforming them into ctsDSR objects.

1. ElementsList

The ElementsList encapsulates a list of objects to be persisted. The EventHandler populates this list as it receives events from the parser. This is a singleton object.

1. Persister

The Persister interface defines the persist() method. UMLPersister implements Persister and uses the Strategy pattern to delegate work to other Persisters. As the name suggests, Persisters are responsible for sending object to the DAO layer to be persisted. In the case of UMLPersister, calls to other Persisters are also ordered because, for example, ObjectClasses must be persisted before ObjectClassRelationships. It is the responsibility of the Persister to verify whether objects already exist or not.

Note: To avoid complexity, list of all persisters have not been added in figure 15 below.

The complete list of Persisters is mentioned below

* 1. PropertyPersister: T o persist Property
  2. PackagePersister: To persist package related information to corresponding domain objects.
  3. DEPersister: To persist DataElements
  4. OcRecPersister: To persist ObjectClassRelationships
  5. DECPersister: To persist DataElementConcepts.
  6. ObjectClassPersister: To persist ObjectClass
  7. ConceptPersister: To persist Concepts and related data structures
  8. ValueDomainPersister: To persist ValueDomain information
  9. RegistrationAuthorityPersister: To persist RegistrationAuthority information
  10. ContactPersister: To persist contact information for submissionContact, stewardshipContact, etc
  11. ClassificationSchemePersister: To persist the Classifications and ClassificationSchemeItems along with their relationships.

1. UMLLoader

UMLLoader is the captain of all classes. UMLLoader is responsible for instantiating the Parsers, Listener, and Persister, thus acting as a factory for those interfaces. It also is responsible for initializing the ElementsList.

1. UML Diagrams

Sequence Diagram

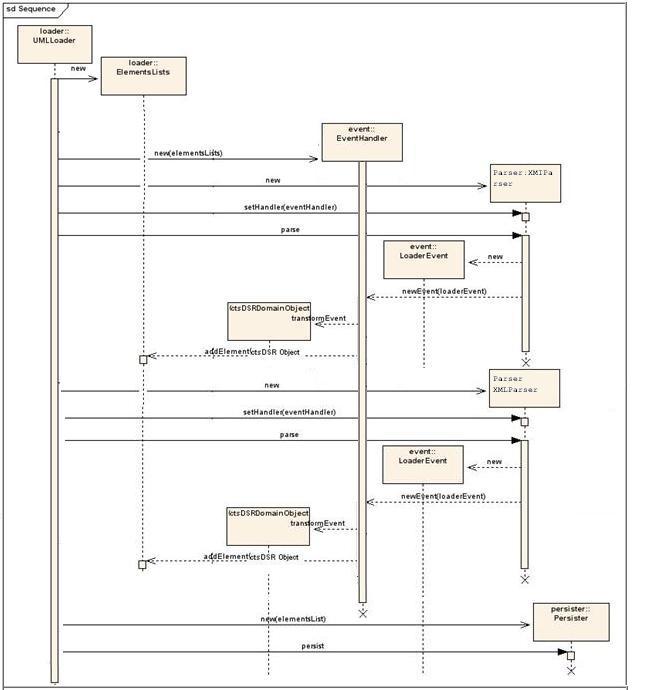


Figure UML Loading Sequence Diagram

Class Diagram

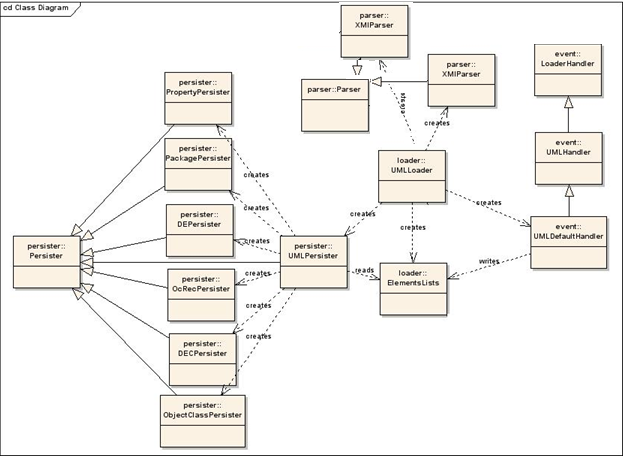


Figure UML Loading Class Diagram

1. Configuration Properties

Certain properties/attributes which are required for storage of metadata are not available from the XMI file. These include the administration and identification data. To capture this data an XML configuration file (ctsDSRConfigurations.xml) needs to be appropriately edited by the model owner. It contains tagged values for each of the attributes required to be stored in the repository and not available in the XMI.

Currently this file contains configurations for the following:

1. Registration Authority
2. Stewardship Contact
3. Submission Contact
4. Organization submitting the model
5. Reference documents
6. Context

The schema for the XML is as depicted in the following diagram.



Figure : ctsDSRProperties.xml

# Glossary

|  |  |
| --- | --- |
| **Term** | **Meaning** |
| POC | Proof Of Concept |
| XMI | XML Metadata Interchange. This is an OMG standard for exchanging metadata information via eXtensible Markup Language (XML). |
| Metadata | Metadata is definitional data that provides information about or documentation of other data managed within an application or environment. |
| DE | This is the feature used by applications like caTissue and caElmir to extend static models and define new models to capture research data |
| ctsDSR | Clinical and Translational Sciences Data Standards Repository- This is the name given to the proposed federated metadata repository |
| administrativeStatus | designation of the status in the administrative process of a registration authority for handling registration  requests |
| registrationStatus | designation of the status in the registration life-cycle of an administered item |
| NCI | National Cancer Institute |
| caDSR | The caDSR is a database and a set of APIs and tools used to create, edit, control, deploy and find common data elements (CDEs) for metadata consumers and for UML model development. The common data elements are developed by the NCI, together with caBIG® partners in the research community |
| cgMDR | The CancerGrid metadata registry software is an open standards, open source implementation of ISO11179-3 allowing users to set up a local metadata registry and populate it with metadata element definitions appropriate to their needs. |

# References

|  |  |
| --- | --- |
| ctsDSR Presentation | <https://gforge.nci.nih.gov/docman/view.php/578/16038/ctsDSR%20Presentation_Final.pptx> |
| ctsDSR Gap Analysis Document | <https://gforge.nci.nih.gov/docman/view.php/578/16036/GapAnalysis-caDSR-cgMDR-DynExt%20Final.doc> |
| ctsDSR Vision And Scope Document | <https://gforge.nci.nih.gov/docman/view.php/578/16037/ctsDSR_Vision_and_Scope_Final.doc> |
| ctsDSR Implementation Plan | <https://gforge.nci.nih.gov/docman/view.php/578/16035/ctsDSR%20Implementation%20Plan%20Final.doc> |
| ctsDSR Model | https://gforge.nci.nih.gov/docman/view.php/578/16277/ctsDSR.eap |
| caCORE Framework | <https://wiki.nci.nih.gov/display/caDSR/Overview+of+caCORE> |
| caCORE SDK | http://ncicb.nci.nih.gov/infrastructure/cacoresdk |
| caGrid Framework | https://cabig-kc.nci.nih.gov/CaGrid/KC/index.php/CaGrid\_Overview |
| caDSR | http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore\_overview/cadsr |
| cgMDR | https://wiki.nci.nih.gov/display/caDSR/cgMDR |
| caBIG | https://cabig.nci.nih.gov/ |

# Appendix

1. XMI to ctsDSR Objects Mapping

ctsDSR closely resembles to the way the XMI data is mapped to the domain objects. The tagged values from the XMI are treated the same way as in the UML Loader for caDSR. Kindly refer the following document for the mappings from UML Model/XMI to ctsDSR.

<https://gforge.nci.nih.gov/docman/view.php/16/15449/caCORE_SIW-UMLLoader_v40_TechnicalGuide.pdf>

1. Kindly refer glossary for the definition of the term. [↑](#footnote-ref-2)
2. Kindly refer glossary for the definition of the term. [↑](#footnote-ref-3)