SDK 4.x CQL to HQL translator dependency on domain model

# Current state

Currently, the CQL2ParameterizedHQL class takes the following parameters in its constructor:

* DomainTypesInformation
  + Provides information on the Java data type of each attribute of each class involved in the service’s data model.
* RoleNameResolver
  + Resolves role names of associations when they are not provided directly in the CQL query – role name is an optional attribute of Association
* ClassDiscriminatorResolver
  + An interface which provides a means to determine the value which must be passed to the special .class property of data types involved in an inheritance hierarchy
* caseInsensitive Boolean
  + Indicates that the generated HQL queries should make use of the lower() operator to force processing of queries without regard to the case of attribute values.

## Domain Types Information

This is a special XML document (deserialized into Axis beans) which contains information about each class and its attributes, and is used to make decisions about what Java data type to create for attribute parameters used in generated HQL queries. This document is generated by the data service creation style and wizard using the domain model as its original source of information.

## Role Name Resolver

This is a utility class which is constructed using the service’s domain model, and uses information contained in it to determine the role name of associations which don’t explicitly indicate one. In the case of non-existent or ambiguous associations (i.e. more than one association between the parent and associated class), it will return an appropriate error.

## Class Discriminator Resolver

This is an *interface* which determines the value which should be used for a given data type’s .class attribute in HQL. A concrete implementation of this interface exists which uses the HBM files generated by the caCORE SDK to determine the discriminator value, but also requires the caGrid domain model to understand class hierarchies.

# Removing the Domain Model dependency

## Domain Types Information

The information made available by the domain types information document and tools is also retrievable from the HBM documents, albeit in a more difficult to reach form. Such a tool would require knowledge of the location of the –orm.jar generated by the SDK, and would require functionality which can discover superclasses of a given data type so it can navigate the inheritance hierarchy.

## Role Name Resolver

This class could be reengineered to look up role names within the HBM files rather than using the domain model. This has the disadvantage that HBM files use a dramatically different syntax for different types of associations, but it’s not an impossible problem to solve given there appear to only be a few variations of the syntax. Such a reengineering would require knowing the location of the –orm.jar file the SDK generates, and then facilities to locate the HBMs for both the source and target of the association. To provide additional flexibility, this class could be abstracted as an interface and a concrete implementation so the functionality it provides can be done so in the most efficient way possible for a given deployment scenario.

## Class Discriminator Resolver

Since this is simply an interface, the facilities it provides can be implemented in any way the caller of the CQL to HQL translator prefers. The concrete implementation which resolves class discriminators from HBM files only uses the domain model for determining the base class of a given data type.

All three of the above tools require the ability to understand class hierarchies and retrieve HBM files based on that information. Currently, they rely on the domain model for this information, however this functionality could be replaced in one of the following ways:

* Use Java’s reflection capabilities to repeatedly call “getSuperclass()” on the class being searched until java.lang.Object is reached. The previous class is the base class within the model
  + The beans jar must be present on the classpath (not a problem for SDK-backed data services)
  + Potentially slow, but class discriminators are cached after they’ve been looked up once
  + Have to handle class not found exceptions and similar issues
* Read all the HBM files either at instantiation or on-demand to locate the class and determine its base class
  + Would have to know the location of the –orm.jar file the SDK generates
  + The SDK uses the convention that the HBM files are placed in a directory structure mirroring the data model’s packages. This helps for base classes, but probably breaks down if a subclass resides in a different package from the base class.

Given the probable difficulties with the second option above, my recommendation is the first route of using the class hierarchy.