Gaudi Version 9

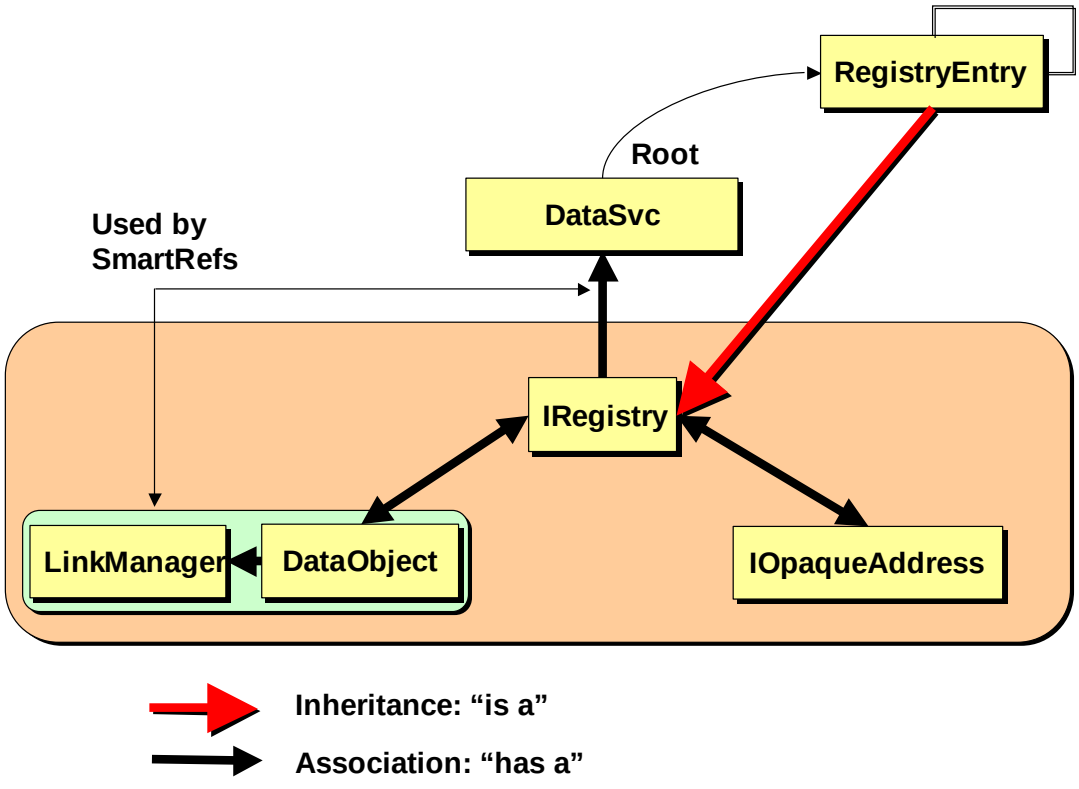
Data Store Redesign

M.Frank
CERN/LHCb

The Cookbook for a Hopefully Painless Upgrade

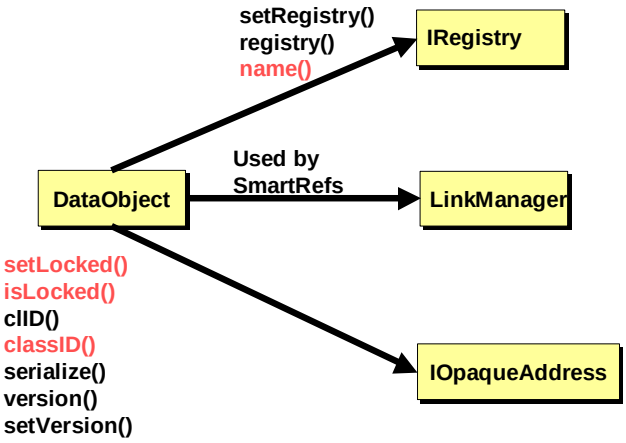
The requirements of the *ATLAS StoreGate* project were incompatible with the existing version of the Gaudi code. For this reason an upgrade of the current Gaudi version was necessary. Unfortunately this upgrade is not compatible with previous versions and changes to user code are very likely. Emphasis was put to minimize changes to physics algorithms. However, information about the internals of the transient data stores, which were accessible through the *DataObject* class are no longer present, but still can be accessed. The purpose of this document is to describe how to access the missing ingredients.

The Transient Datastore



Any access to data (Event data, Detector data, histograms etc.) in the data store is through the “store keeper”, the DataSvc object. The store is structured by objects of type RegistryEntry, which allow building a tree like data structure. The public part of these entries is specified through the interface IRegistry. Any data item can be represented by the persistent address (interface IOpaqueAddress) or the transient representation of the object itself: Saving an object yields an opaque address, and a valid address allows retrieving the transient representation of the object.
It is expected that the changes affect code, which deals with the transformation of objects between the transient representation and the other representation.

Given a *DataObject* instance, the following diagram is visible to the programmer:



The changes typically affect the handling of

- Object references using the *SmartRef* mechanism
- Access of directory leaves of the next hierarchical layer in the tree.
- Registration of new directory leafs by converters.

Simple code translation recipes:

Typical string information to be retrieved from a DataObject

```
DataObject* pObj = <valid and registered data object>;
```

Gaudi v8	**Gaudi v9**
`pObj->localPath()`	`pObj->registry()->name()`
`pObj->fullpath()`	`pObj->registry()->identifier()`
`pObj->location()`	***does no longer exist. Use:***
	`std::string& f=pObj->registry()->identifier()`
	`std::string location = f.substr(0, f.rfind('/'))`

Access to the parent of a DataObject in the data store

In Gaudi v8 the access to a parent object was simply given by the DataObject itself:

```
DataObject* someObject = <some valid and registered object>
DataObject* parent = someObject->parent();
```

In Gaudi v9 the tree structure and hence access to the parent is no longer directly exposed. The access is possible using the IdataManager interface of the transient data store:

```
DataObject* someObject = <some valid and registered object>
SmartIF<IdataManagerSvc> dataMgr(someObject->registry()->dataSvc());
If ( dataMgr )  {
  IRegistry* pRegParent = 0;
  StatusCode status = dataMgr->objectParent(someObject, pRegParent);
  if ( status.isSuccess() )    {
      DataObject* parent = pRegParent->object();
```

Access to the tree structure of the data in the data store

The tree structure of the store is no longer accessible through the *DataObject* itself. Code like:

```
DataObject* pObj . . .  <valid and registered DataObject>
for( DataObject::DirIterator i = pObj->dirBegin(),
                              iend = pObj->dirEnd(); i != iend; i++ )  {
  std::cout << (*i)->fullpath() << std::endl;
}
```

will no longer work. The access to the child leaves must be done through the IDataManagerSvc interface of the data service:

```
DataObject* pObj . . .  <valid and registered DataObject>
SmartIF<IdataManagerSvc> dataMgr(pObj->registry()->dataSvc());
If ( dataMgr )  {
  typedef std::vector<IRegistry*> Leaves;
  Leaves leaves;
  IRegistry* parent = 0;
  StatusCode status = dataMgr->objectLeafs(pObj, leaves);
  if ( status.isSuccess() )    {  … now work with object leaves …
    for( Leaves::iterator i = leaves.begin(),
                          iend = leaves.end(); i != iend; i++ )   {
    std::cout << (*i)->identifier() << std::endl;
  }
}
```

Information retrieved from an IOpaqueAddress/GenericAddress

```
IOpauqeAddress* pAdd = <valid opaque object address>;
```

Gaudi v8	**Gaudi v9**

pAdd->dbName() and pAdd->containerName()	**Removed because of logical inconsistencies. Use:**
	const std::string* string_params = pAdd->pars();
	const std::string& dbName = string_params[0];
	const std::string& cntName = string_params[1];

pAdd->objectName()	was redundant and useless. The object name must be supplied when a new address is registered as a leaf in the data store.

pAdd->genericLink()->…	**no direct equivalent. Use to access/set integers:**
	const unsigned long* int_params = pAdd->ipar();
	Note, that the interpretation of these integers depends on the technology used. A typical mapping is:
	unsigned long link_id = int_params[0];
	unsigned long record_id = int_params[1];

Note that the content of pAdd->pars() and pAdd->ipars() depends on the technology used and hence on the concrete implementation of the IOpaqueAddress object. In the most general case the returned pointers are not necessarily defined.

Typical Situations affected by the changes

Creating a Valid SmartRef Object inside a Converter

SmartRef objects are used to implement associations between objects residing in the transient data store. Any *SmartRef* has several ingredients, which must be validated. These are:

1. The pointer of the owner of the smart reference. This pointer must be valid, what typically is not a problem, because it is the pointer of the object created in the converter.
2. The identifier of the link within the *DataObject*.
3. The key to the target object
4. The pointer to the target object. If this pointer is invalid, the key must be valid, because this key allows retrieving a valid pointer on demand.

The link identifier together with the object key (2+3) has the same information content as the pointer itself. A valid pointer only avoids an additional lookup in the data store and hence leads to better performance. Up to Gaudi v8 this was typically implemented the following way:

```
IDataDirectory* pDir = pAddress->directory()->parent();
std::string path = pDir->fullpath() + "/MCVertices";
dataProvider()->findObject(pDir, "/MCVertices", (DataObject*&)pVtx);
long id = particles->addLink(path, pVtx);
. . .
MCParticle* p = new MCParticle();
MCVertex*   v = (pVtx && idx>=0) ? (*pVtx)[idx]:0;
p->setOriginMCVertex( SmartRef<MCVertex>(p, id, idx, v) );
            … where                  p   = Owner of SmartRef
                                     id  = Link number
                                     idx = Target object's key
                                     v   = Target pointer if availible
```

Due to the changing implementations of *IRegistry* and *DataObject*, this must change to the following:

```
IRegistry* pParent = 0;
status = m_dataMgr->objectParent(pDir, pParent);
if ( status.isSuccess() )   {
  ObjectVector<MCVertex>* pVtx = 0;
  std::string path = pParent->identifier()+"/MCVertices";
  dataProvider()->findObject(pDir, "/MCVertices", (DataObject*&)pVtx);
  long id = particles->linkMgr()->addLink(path, pVtx);
      . . . Rest as above . . .
```

Registering Leaf Addresses within a Converter

In Gaudi v8 and earlier, there was no clear recipe specified to register the address of a leaf object from within a converter object. This typically lead to ugly hacks directly manipulating the registry entry itself using the *dynamic_cast<>()* operator. This can now

be done using the *addressCreator()* accessor, which is present in any converter and the data manager interface:

```
IopaqueAddress* pA = 0;
DataObject* pObj . . .  <valid and registered DataObject>
iret = addressCreator()->createAddress( <storage type of new address>,
                                        <class id of object>,
                                        <string params (std::string*)>,
                                        <integer params (long*)>,
                                        pA);  // Resulting address
if ( iret.isSuccess() )   {
  SmartIF<IdataManagerSvc> dataMgr(pObj->registry()->dataSvc());
  If ( dataMgr )   {
    iret = dataMgr->registerAddress(pObj, "/Tracks", pA);
    if ( !iret.isSuccess() )   {
      pA->release();
    }
  }
}
```

In order to fulfill it's job, the converter is instrumented with a set of services, as it can be deduced from the class diagrams shown in Figure 1 and Figure 2. These services can be accessed using the following accessors:

- Get Data provider service
  `virtual IDataProviderSvc* dataProvider()    const;`
- Get conversion service the converter is connected to
  `virtual IConversionSvc* conversionSvc()    const;`
- Retrieve address creator facility
  `virtual IAddressCreator* addressCreator()   const;`

Please note, that neither the *ConversionSvc* nor any converter does interact with any of these services itself. These services exist solely for the use of the implementer, access to these services ease life.

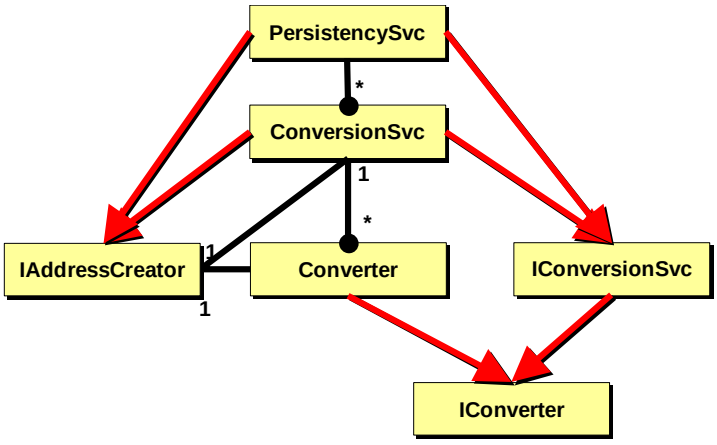


Figure 1 The class diagram of the persistency mechanism.

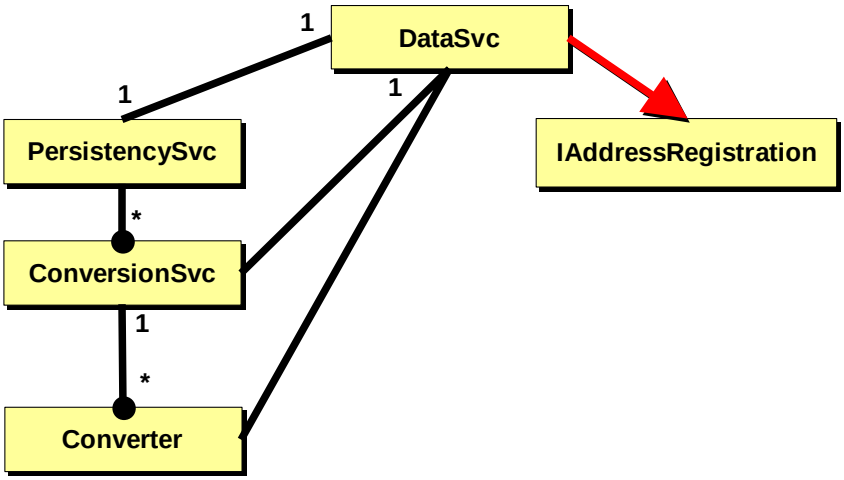


Figure 2 The usage of the data service by the conversion mechanism. The link of the converters to the data service is only a use link, not mandatory.

Create Persistent Objects using the ConversionSvc

In Gaudi v8 the *ConversionSvc* class accepted only one call to

- Create transient objects from their persistent representation (*createObj*(…))
- Save objects to its persistent representation *(createReps(…))*.

Data conversion is a two-step process: first the object data must be converted, and then the references to other objects can be filled. Otherwise it is impossible to handle bi-directional pointers. In addition, after the creation of a transient object, the object registry must be updated.

In Gaudi v8, the *ConversionSvc* handled object creation, update of the registry and filling of the references internally. As a consequence, a callback to the data service was necessary and coupled the use of the *ConversionSvc* tightly to the data store service. This causes several problems if object conversion is required without a data store: e.g. in the way *StoreGate* or the *DetectorDataSvc* use the *ConversionSvc*.

In Gaudi v9 this coupling was removed. However, any clients must itself specify the three steps as follows:

- When creating the transient representation the clients of a *ConversionSvc* object must first invoke the object creation and then the validation of the references. What was originally:

  ```
  status = m_dataLoader->createObj(pAddress, pObject);
  ```

  must become now:

  ```
  status = m_dataLoader->createObj(pAddress, pObject);
  if ( status.isSuccess() )    {
        . . . UPDATE Registry . . .
        status = pLoader->fillObjRefs(pAddress, pObject);
            . . .
  ```

- The same calling sequence in a fully symmetric way must be applied when objects are written to a persistent medium. Clearly, if the conversion process must be specified in two steps it does no longer make sense to act on several objects at once. Old code:

```
IDataSelector* selectedObjects = . . .;
status = m_pConversionSvc->createReps ( selectedObjects );
```

Replacement code:

```
  IOpaqueAddress* pAddress=0;
  IDataSelector* sel=. . .;
  StatusCode iret;
  IDataSelector::iterator j;
  for ( j = sel->begin(); j != sel->end(); j++ )    {
    iret=m_pConversionSvc->createRep(*j, pAddress);
    if ( iret.isSuccess() )      {
      . . . UPDATE Registry . . .
    }
  }
  for ( j = sel->begin(); j != sel->end(); j++ )    {
    IRegistry* pReg=(*j)->registry();
    iret=m_pConversionSvc->fillRepRefs(pReg->address(), *j );
    if ( !iret.isSuccess() )      {
      status = iret;
    }
  }
}
```

In both conversions the update of the registry cannot be done inside the conversion service, because this service has now no collaboration with the data service anymore. The client must do it!