



# The StoreGate

## A Data Model for the ATLAS Software Architecture

The EDM Working Group  
Editors: P. Calafiura, S. Rajagopalan

With the successful adoption of the Gaudi framework<sup>1</sup> in May 2000 as a starting point for the ATLAS software development, the effort is now focussed on understanding the specific ATLAS requirements and ensuring a framework design that will satisfy them. The Event Data Model (EDM) is a crucial element of the overall infrastructure that defines the management and use of data objects in its transient state. Based on the experience in using the Gaudi infrastructure and subsequent discussions within the EDM working group, we have developed a prototype design (StoreGate). We have implemented and tested some of the proposed design features stressing on the interface to the client's access to data objects while currently maintaining the underlying Gaudi infrastructure. This note describes the proposed elements of the StoreGate design and the prototype implementation.

### Motivation

The ATLAS software architecture belongs to the blackboard family: data objects produced by knowledge objects (e.g. reconstruction modules) are posted to a common in-memory database from where other modules can access them and produce new data objects. This model greatly reduces the coupling between knowledge objects containing the algorithmic code for analysis and reconstruction - a knowledge object does not need to know which specific module can produce the information it needs, nor which protocol it must use to obtain it<sup>2</sup>. Algorithmic code is known to be the least stable component of HEP software systems and the blackboard approach has been very effective at reducing the impact of this instability, from the Zebra system of the Fortran days to the InfoBus Java components architecture.

The trade-off for the data/knowledge objects separation is that knowledge objects have to identify data objects they want to post or retrieve from the blackboard. It is crucial to develop a data model optimized for the required access patterns and yet flexible enough to accommodate the unexpected ones.

Starting from the recent Event Data Model workshops and discussions<sup>3</sup> as well as from the experience of other HENP systems such as BaBar, CDF, CLEO, DØ, we identified some requirements that would not be readily satisfied using the existing Gaudi Event Data Model:

- Identify (collections of) Data Objects based on their type.

---

<sup>1</sup> for details, refer to : <http://lhcb.cern.ch/computing/Components/html/GaudiMain.html>

<sup>2</sup> the "interface explosion" problem described in component software systems

<sup>3</sup> <http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture>



- Identify (collections of) Data Objects based on the identifier of the algorithm which added them to the Transient Data Store (TDS).
- A scheme that allows developers optionally to define key classes tailored to their Data Objects and to use the keys to store and retrieve the objects from the TDS.
- A well-defined and supported access control policy: as will be discussed later on, objects stored into the TDS should be “almost read-only”.
- A mechanism to hide as much as possible the details of the TDS access from the algorithmic code, using the standard C++ iterator and/or pointer syntax.
- The same mechanism should be used to express associations among Data Objects.
- A mechanism to group related Data Objects into a developer-defined view that provides a high-level alternative access scheme to the store.
- A mechanism to define a flexible “cache-fault policy” (that is to say a way to create an object requested by the user and not yet in the TDS). This should include the functionality to reconstruct a Data Object on demand.

## System Features and Entities

- A **Store Gate** will allow algorithmic code to interact with the TDS.
- When an Algorithm invokes the Store Gate retrieve method, it is returned a **Data Handle** or a list of Data Handles to the collection in the Transient Store. The Data Handle defines the interface to access the Data Objects retrieved. From the user perspective a DataHandle behaves as a C++ pointer/iterator, and its first implementation pointer is indeed a C++ const pointer. It will have to evolve into a more complex object to satisfy the requirements mentioned above. A complete implementation of the DataHandle should include:
  - Begin/end methods providing iterators over contained objects,
  - An “almost const” access policy that allows, for example, to add elements to a collection and mark a Data Object as “to be written on persistent store”<sup>4</sup>.
  - An update method used (presumably by the TDS) to mark the current Data Object the DataHandle refers to as “obsolete” when, for example, a new event is read in.
  - Support for persistable object associations (e.g. track → associatedHits),
  - The ability for the client to view the data in different ways.
- A user-defined **Key** can be used while registering/retrieving the DataObject. Any class that provides an ordering (e.g. “greater than”) operation can be used as a Key, for example an Atlas detector element identifier, an algorithm instance identifier<sup>5</sup> or a string (a-la-Gaudi). In the prototype, the key object is converted into a string<sup>6</sup> that is added to the Event Data Service object “pathname”.
- A user-defined Selector can be provided that can filter on the contents of the collection or contained objects in the collection.

---

<sup>4</sup> the debate about the access policy is still open, see section C.2 (Store Access Policy) for more details

<sup>5</sup> the concept is derived from DØ RCPID

<sup>6</sup> and therefore a Key class must also provide methods to convert to/from a string



- A virtual **Proxy** mechanism is used to represent Data Object instances that are not yet in the TDS. The Proxy will define the procedure to create these instances, either by reading them from persistent storage or by reconstructing them on demand. The Proxy can also be used, in conjunction with the DataHandle, to implement, if required, complex access control policies for the Data Objects.

## Characteristics of the Transient Data Store (TDS)

All objects in the transient data store will inherit from a common base class (DataObject). The objects in TDS will support storage of either collections (of several contained objects) or single objects. It is however assumed that in most instances, clients will store collections in the Transient Store; for example a collection of Track Objects (TrackCollection) or a collection of electron candidates. A collection will contain objects of only one type. However, several types of objects can inherit from a base-type and be stored in the collection as the base-type. Multiple instances of the same collection or objects can be registered with the TDS.

Each collection or object in the TDS will be associated with a History object. The History object will contain specific information on which Algorithms made the object, the configuration of these algorithms as defined by some run control input file, environmental information such as calibration and alignment information that made the collection. The purpose of the History object is two-fold: it will precisely define how the collection was made such that it is reproducible at a later stage, and the client can select on the basis of the information in the History Object. An example of the latter is as follows: “Clusters in the EM calorimeter are found in two different ways in the reconstruction process, using a cone algorithm and a nearest neighbor algorithm. In each instance, cluster objects are stored in a ClusterCollection. Hence there will be two ClusterCollections (same type) in the TDS each simple reflecting a different algorithm. Since the History Object carries this information, a downstream client will be able to request EM cluster made using a specific algorithm.”

The TDS will contain data objects that are either persistable or required for communicating between algorithms. An almost const-access policy is being proposed for accessing data objects in the Transient Store. See detailed discussion on this subject and how such policy can be implemented in section C.2 (“Store Access Policy”).

Tagging an object in the TDS by a downstream client will be forbidden. An example of a tag may be to simply mark it as having been used for the convenience of a single client or associating it with objects which have been produced at a later time in the reconstruction chain (so called forward pointing of objects). That is a "hit object" cannot be explicitly tagged as used or modified to reflect which track (constructed later in time) it is associated with. This is simply because the same hit collection may be used by other clients for whom such tagging may be irrelevant. However tagging or forward pointing of DataObjects must be supported in other ways such as with association objects<sup>7</sup>.

---

<sup>7</sup> Refer to discussion in Section E: “Inter Object Relationships”



## Detailed Design Discussions

We provide detailed discussions on the following issues:

- A) Store Gate and StoreGateSvc
- B) Keys
- C) DataHandles
- D) Selectors
- E) Inter Object Relationships
- F) Data Proxies

### **A) Store Gate and StoreGateSvc**

The Store Gate defines the interface through which the algorithmic code will access the TDS. The Store Gate will extend the existing Gaudi TDS, providing the extra functionality mentioned before: type-safe accesses, flexible indexing mechanism, support for access control and user defined data-views. Since in the medium term the Store Gate may replace or be merged with the existing Gaudi EventDataSvc, it is important to maintain compatibility with the current interface. In the prototype a new Gaudi Service, the StoreGateSvc provides templated methods to register and retrieve Data Objects by type. It is implemented as a layer over the existing EventDataSvc.

#### A.1 Storing in Transient Store by Types using StoreGateSvc

The register interface would use templated methods parameterized by "Collection" types and would accept a pointer to the collection (ObjectVector), convert it into a pathname that Gaudi requires and store it in the Gaudi TDS.

The path name would look like: "*prefix/typename/key*" where:

- The default *prefix* is *"/Event/Store"* but allowing for a user pathname to be specified. While the proposal is to store processed collections/objects flat under *"/Event/Store"*, the system allows the option of maintaining a hierarchical structure should one choose to do so.
- The *typename* would be determined from the pointer to the Collection. Note that *"typeid.name()"* to access the type information is not portable across platforms. Other options to obtain the type information are being explored.
- The *key* is user-defined and unique to the container. The key is *optional*. Since the key is converted to a string and forms a part of the pathname, it provides an additional degree of hierarchy.
- The system will provide a unique id (unique within an Event) for the container. This could be unique for every container within an event – or unique amongst each type of container.



The proposed declaration of the register methods in StoreGateSvc is:

```
Class StoreGateSvc {
    public:

        // Simple type storage
        StatusCode registerObject<T>(T* collection);
        // Store by Key – See discussion of Key for syntax explanation
        StatusCode registerObject<T>(T* collection, typename T::Key& key);
        // Hierarchical storage
        StatusCode registerObject<T>(string& pathname, T* collection);

    private:
        // Define default pathname for Transient Store (“/Event/Store”)
        string m_pathname;
}
```

## A.2 Type Safe Access to DataObjects using the StoreGateSvc

The StoreGateSvc provides retrieval methods for access of DataObjects in the Transient Store by types.

- The retrieve method returns a DataHandle to the collection. The Data Handle could be a ‘const’ pointer.
- The retrieve method allows keyed access if the DataObject is stored with a Key.
- The retrieve method allows keyless access whether or not the DataObject is stored with a key.
- The retrieve method also provides a mechanism to return a collection of DataHandles. If only one collection is requested, the most recently added collection of the requested type is returned to the client.
- It will also provide a mechanism to retrieve Data Objects as its superclass. For example, if a collection MuonTrackCollection inherits from TrackCollection, the collection of tracks can be accessed as type TrackCollection or MuonTrackCollection. This avoids the client to access DataObjects by only one type and have the burden of performing a dynamic cast operation.

See discussion on DataHandles for additional functionality of and access policy to the returned object.

Since DataObjects are stored as “prefix/typename/key”, the type information specified in the templated retrieve method is converted to a string to retrieve collections from the Gaudi transient store. If no key is specified, the requested type is first matched with possible collections in “/prefix/typename”. If none are found, all possible matches among “/prefix/typename/key” are made. Either one or all the matching collections are returned depending on the request.



The proposed declaration of the retrieve methods in storeGateSvc is:

```
Class StoreGate {  
  
    public:  
        // Return the most recent DataHandle based on type (See discussion on DataHandles)  
        StatusCode retrieveObject<T>(DataHandle<T>& handle);  
  
        // Return a Datahandle based on key information (See discussion on Key)  
        StatusCode retrieveObject<T>(DataHandle<T>& handle, typename T::Key& key);  
  
        // Return a Datahandle based on a user specified selection criteria  
        // This could be a part of the Datahandle (See discussion on Selectors and Handles)  
        StatusCode retrieveObject<T>(DataHandle<T>&handle, Iselector& selector);  
  
        // Return a Datahandle by specifying the unique system ID  
        StatusCode retrieveObject<T>(unique_system_assigned_container_id);  
  
        // Return a list of Datahandles  
        StatusCode retrieveAllObjects<T>(std::vector<DataHandles<T>> list_handles);  
}
```

### A.3 Supporting Multiple Client Views of the Data

In the introduction we mentioned how client code may identify an object in the store providing its type, its “maker” or some other indexing mechanism, such as its ATLAS detector element ID<sup>8</sup>. We can predict that some indexing mechanism or **Namespace**, in particular the “Canonical” one of object Type & Instance, will be widely used and therefore must be supported efficiently. We believe this is the case for the prototype StoreGateSvc we have developed. The prototype allows identifying an instance using a user-defined key but does not support a “wildcard” mechanism on that key. Therefore while you are allowed to request all the object of type LArCellContainer from the store you can’t request all the raw-data objects in the LAr branch of the detector element tree.

### A.4 Multipletons & Templated StoreGate

We discussed the possibility of a templated StoreGate class instead of templated methods as described above. A templated StoreGate class (instead of templated methods) has several advantages:

- Templated methods are in general not supported by all compilers.

---

<sup>8</sup> <http://atlas.web.cern.ch/Atlas/GROUPS/DATABASE/Documentation/documentation.html>



- It is not possible to define virtual templated methods. Hence the templated register/retrieve methods cannot be defined in a base class and overloaded in a subclass.
- Using templated methods is rather cumbersome. The method implementation must be included in the client code.
- The use of several StoreGate classes (one for each type T) eases the retrieval of objects. One does not have to loop over all objects in the TDS to determine if the requested type is available in the TDS or not. Look up and browsing DataObjects of the same type in the TDS is simplified with the use of a templated StoreGate Class.

The question arose on how the framework could know apriori the value of the parameter type T and instantiate StoreGate<T>. The concept of multipleton was proposed in this context. A multipleton is simply a collection of singletons. In this case, each specific instance of StoreGate<T> would be a singleton, the collection of singletons managed by the framework. This approach would instantiate a StoreGate<T> if it already does not exist when the client requests its pointer.

## **B) Keys**

The collections in the Transient store could be associated with a Key object when registered and later retrieved. After considerable discussions we came to the following conclusion on what the Key is and its possible usage:

- The Key object must be user defined. It could be an Identifier of some kind, a string or anything that a client may choose to use to tag a collection. It must inherit from a base class AbsKey.
- Each DataObject class defines its “natural” Key type and a factory method to build instances of it. This promotes the use of a single Key class for each container and the downstream client can know the key used to tag the DataObject. The alternative is to not couple the Key object with the DataObject and allow any Key (which is a subclass of AbsKey) to be used. In the prototype implementation, the Key has been coupled to the DataObject.
- In the prototype implementation, the Key would be converted to a string to form a part of the pathname. The Key class must therefore provide a getString() method that registerObject will use while storing the object in the transient store.
- A keyless access to a DataObject must be possible, even if the DataObject was registered with a key.

Here is an example of a Key class implementation for a DataObject Collection “TrackCollection”:

```
Class TrackCollection {  
  
    class TrackCollectionKey;
```



```
typedef      TrackCollectionKey  Key;

    ....
}

// The class mykey: (inherits from a base class AbsKey)

class TrackCollectionKey : public AbsKey {

    public:
        TrackCollectionKey(Identifier id);
        ~TrackCollectionKey();
        string getString();
        Identifier decodeString();

    private:
        Identifier m_id;           // The identifier
        string m_string;         // The associated string
}

// The client use while registering objects:
    Identifier id = atlas_id.pixel();           // returns identifier for pixels
    TrackCollectionKey mykey(id);             // form a key with the identifier

// register your object with the key:
    storeGateSvc() → registerObject(TrackCollectionPointer, mykey);
```

The registerObject method in StoreGateSvc() accepts a “T::Key key” forcing the client to use a Key defined in MyCollection (using typedef). A possible alternative is to not enforce this requirement (in which case the registerObject method in StoreGateSvc() will accept the key as the base class type AbsKey). The registerObject would then get the string associated with the Key by invoking the getString method. Hence, the Key object must provide a getString method (enforced through the base class AbsKey). The decodeString method is optional. The retrieveObject works similarly.

Note further that the MyCollectionKey example described above accepts an Identifier. The user (or actually the developer of the MyCollection class) can define other IDs, strings as keys or possibly make use of multiple inputs. The only requirement is to provide a getString() method.

### **C) Data Handles**

A Data handle must provide a natural, almost trivial, interface for the client code:

```
// Create a Data handle
    DataHandle<TrackCollection> myhandle;
```



```
// Pass to retrieve which loads the DataHandle and returns it
    StatusCode sc = storeGateSvc() → retrieveObject(myhandle);

// Use it like any pointer . DataHandle points to a DataObject
    myhandle → use_me();
```

## C.1 List of Data Handles

In the example above note that myhandle points to a *single* collection of tracks – probably the last TrackCollection that was registered to the Transient Store. If there are several track collections that satisfy the selection requirements, and the client wants to browse through all of them, you could imagine doing:

```
list<DataHandle<TrackCollection>> myHandleList;
StatusCode sc = storeGateSvc() → retrieveObject(myHandleList);
TrackCollection::iterator trackColl = myHandleList.begin()
for ( ; trackColl != myHandleList.end(); trackColl++)
{
    ... //your stuff here
}
```

One could instead provide a DataHandleList class which is invoked by the client as:

```
DataHandleList<TrackCollection> myHandleList;
```

MyHandleList can be passed to the retrieveObject in just the same way as “myHandle”. This has not only the effect of making the client code look more simple and manageable but allows us the possibility to introduce additional features in DataHandleList – while hiding these from the client applications.

## C.2 Store Access Policy

Should a client be allowed to modify a DataObject in the store using a DataHandle returned by retrieveObject? More in general, what is the access policy for DataObjects in the TDS, and how is it enforced? Let us review the strategy adopted by other experiments’ EDMs. In BaBar the policy is that once an object is committed to the store it should not be modified anymore. On the other hand this policy is not enforced neither at compile nor at run time by any tool, so exceptions are possible. The DØ approach is to pass back a handle that behaves like a const pointer. Modifications to objects in the store are made only to a copy of the object. CDF adopts a similar policy but the handle allows adding information to a Data Object, in particular by inserting new elements to a collection retrieved from the store.

While a const-access may seem desirable, such a policy would be extremely inefficient in some instances. For example, a package that is responsible for producing a collection of



objects may have several sub-algorithms that perform different functions and modify the contents of these objects. A const-access policy would force passing around these objects between the sub-algorithms.

ATLAS is trying to define an access policy that is safe as the DØ approach but somewhat less restrictive. One possibility is to follow the CDF idea of an “almost const” access. Several strategies have been discussed on how to implement such a policy, which are briefly outlined below:

- The DataHandle would provide a const pointer “passthrough” accessor and a limited set of non-const methods that would allow for example to add an element to a container or to flag a DataObject as “to-be-persisted”. The problem with this approach is the physical coupling created by the non-const methods. Unless the requirements are well understood from the beginning each time a new non-const method is added most of the application code will need to be recompiled and linked (not to mention tested).
- A private store can be established for each Algorithm where the Algorithm (and the associated sub-algorithms) can modify the objects in this store. Other Algorithms may not modify these private objects. Once the objects have been defined, they may be committed to the public TDS. A commit would be equivalent of a Unix "move" operation. A copy of the object in the private store will not exist once the object has been moved.
- A locking mechanism can be established. Until locked, the data object in the TDS can be modified. It would be the algorithms responsibility to ensure that the DataObject is locked to prevent other algorithms from modifying it. Locked DataObjects may not be unlocked. Additional constraints can be imposed if necessary such as: Only locked objects can be made persistent and/or warning log messages about unlocked objects in the store at the end of the event processing.
- The most flexible, if complex, would be to attach to DataObjects an Access Control List not unlike a file in AFS. Algorithms would have to be authorized (by the DataObject maintainers?), before they could modify a given object in the TDS. By default, the DataObject would be read-only.

Other variations on the above themes are entirely possible. We propose for ATLAS that:

- We should allow some level of modifications to the objects at least by algorithms within a package.
- Other packages would only have a const access. They must make a copy of the DataObject, modify it, and register it again (as the same type). This would reflect the new state of the DataObject in the TDS.



- The TDS would contain persistable objects or those that are required for communication between packages and will not be a scratchpad.
- The explicit tagging of an object or associating an object with another that has been reconstructed at a later time will be forbidden. That is a "hit object" cannot be explicitly or associated with a track (which has been made using the hits at a *later time* in the reconstruction). This is simply because the same hit collection may be used by other clients for whom such tagging may be irrelevant. However tagging or forward pointing of DataObjects must be supported in other ways such as with use of association objects. See discussions on Inter-Object relationships.

### C.3 Iterators associated with Data Handles

There are two types of iterators – an iterator over contained objects in the collection that the DataHandle points to, and an iterator over the collections themselves if dealing with DataHandleList. A begin and end method can be provided in both DataHandle and DataHandleList that return an iterator over the respective collections.

- DataHandle.begin() and DataHandle.end() will return the begin and end iterators over the contained objects in the collection.
- DataHandleList.begin() and DataHandleList.end() will return the begin and end iterators over the list of DataHandles (list of DataObjects).

In the section of “Selectors”, the concept of returning only contained objects passing a selection condition will be discussed. This avoids the client having to loop over all the contained objects and performing their own selection of the contained objects.

Note further that the Transient Data Store may contain either Collection objects or single objects, the latter being far less likely but nevertheless is a design requirement. In these instance, where the DataHandle points to the single object that is retrieved from the Transient Data Store, an operation *DataHandle.begin()* and *DataHandle.end()* does not make sense. A possible solution to this problem is to add the necessary intelligence to DataHandle such that an attempted iteration over the single object would iterate over itself. There are other possible solutions that are being discussed.

### C.4 Selectors associated with Data Handle

Selectors associated with Data Handles (see also the discussion on Selectors in the next section). A client may provide a selector to specify the collection he is interested in. For example, the selector may require two highest  $E_T$  EM clusters with a minimum cut of 20 GeV, a typical physics requirement. The cluster normally will be in some collection called ClusterCollection. Let us suppose that an event has 5 reconstructed EM clusters 40, 35, 25, 18, 15 GeV respectively. The first requirement is 2 clusters. This condition can be determined by simply looking at the size of the collection and not knowing anything



about the contents of the cluster objects. The second requirement requires you to browse through the objects in the collection and pick the clusters with 40 and 35 GeV. A preferred way of doing this would be:

```
// Create an instance of your selector
MyEMSelector      sel;

// Pass it to the Data handle
DataHandle<EMClusterCollection> myhandle(sel);

// Retrieve the collection from the transient store
StatusCode sc = storeGateSvc() → retrieveObject(myhandle);
```

“myhandle” now points to a collection : the collection that contains all 5 contained objects. The next natural step for the client is to loop over the contained objects in this collection using `DataHandle.begin()` and `DataHandle.end()`. The begin and end methods could be smarter (having known your selector) and return the iterator over ONLY contained objects that have passed your selection. In this case only two `ContainedObjects` will be returned to the client. The client does not need to repeat the selection by looping through all the 5 objects in the collection again. It was felt that such a selection could be done by clever use of STL algorithms, hence the exact nature of such an implementation is being worked out. See discussion on Selectors in the next section.

## C.5 AutoHandles

An `AutoHandle` is defined as a pointer to an object that automatically updates when the object goes out of the scope of its defined validity. An `AutoHandle` is initialized only once during a job unlike a `DataHandle`. When dereferencing an `AutoHandle`, the validity of the object is checked and a new instance loaded if required. The validity could be as simple as the validity of the Event (in which case the `AutoHandle` points to a new instance of the `DataObject` that is relevant for that Event) or it could be more complex like a time interval or a range of run numbers. Such a capability is attractive especially while dealing with calibration constants, since the client does not have to worry about whether the calibration data being used is valid for the reconstruction or analysis being performed. However it is expected that a `DataHandle` will be used while referring to objects whose context changes from event to event and where the client would want more control. Should the *selector* be associated with the `DataHandle`? It may be necessary to associate it with the `Autohandle` if automatic selection needs to be performed when a new `DataObject` is loaded. See the discussion on selectors in the next section for more details.

## D) Selectors

The user can provide any number of selectors - algorithms that perform a selection based on the information in the collection. There are two types of Selection functions. One that selects collection objects, the other that selects contained objects within the selection. For



example, a selector could request for a collection that has at least two electromagnetic clusters in the barrel region above 20 GeV. A Selector could also select on other parameters such as returning a TrackCollection where tracks were fitted using Algorithm A (instead of Algorithm B), or returning a TrackCollection of given ID. Such selection methods have proved to be extremely powerful and useful in DØ.

The client-defined Selector will inherit from a base class (ISelector), templated with type T, and provide a *match* method accepting T\* (the pointer to the collection object) as input: ISelector will have an implementation like:

```
class ISelector<T> {  
  
    virtual bool match(T* t) = 0;  
};
```

The client selector header file would be as follows:

```
class MySelector :: public ISelector<TrackContainer>  
{  
    public:  
  
    MySelector();  
    // The client selection function receives the pointer to the TrackCollection  
    bool match(TrackContainer* t)  
    ...  
};
```

The usage of the selection (MySelector) in the client code would look like:

```
// Make a Selection Object  
MySelector sel;  
  
// Create a DataHandle and pass it the selector  
DataHandle<TrackContainer> myhandle(sel);  
  
// Retrieve the pointer to the Collection that passes the Selection criteria  
StatusCode sc = storeGateSvc() → retrieveObject(myhandle);
```

The *myhandle* that is returned to the client after having performed the selection is a pointer to the collection object (as always). The predictable next step in the client code would be to loop over the contained objects in the collection (that *myhandle* points to) and perform a further selection over the contained objects in the collection. This step can be completely circumvented by having a second type of Selector (a Contained Object Selector) that accepts the pointer to the contained object in the selection and hiding the loop over contained objects from the client. The client iterates over the contained objects in the collection by getting a pair of iterators from the data handle (using



*myhandle.begin()* and *myhandle.end()* as explained in the previous section). The data handle would perform the loop over the contained objects and pass back to the client only objects that pass the Contained Object Selector.

## **E) Inter Object Relationships**

In this section we will discuss associations between objects: how they are maintained and how the client uses them. To be concrete, we will discuss the classic relation between tracks and the hits. As each track is constructed from a collection of hits, the obvious use case is that the client must be able to navigate from the track to its associated hits. A simple list of bare C++ pointers introduces an undesired physical coupling between DataObjects. They also do not suffice from the persistency perspective: Track and Hit objects may be stored separately and only the track objects may be read back in at a later analysis stage. The C++ pointers to the hits that the track objects carry are no longer meaningful. Hence, we need some kind of smart pointer which when dereferenced retrieves the hit objects on behalf of the track client.

Another interesting use case that has been discussed is the “reverse” navigation from the associated hit object back<sup>9</sup> to the track object. Naively one could hope to implement this association by adding to the hit class a (smart) pointer to the class to be filled after the track has been constructed. But this solution is riddled with problems such as:

- A hit object may not be associated with any track, and the track back pointer would be left undefined.
- A hit object may not be modifiable, for example because it is read from a stream we have no write access to.
- Two track objects may occasionally share a hit object.
- Likewise, several tracking algorithms, all reconstructing the same set of tracks in different ways, may use the same hit object. To which track does the hit belong?

As a third related use case, the client constructing tracks may wish to mark the hit objects as having been used, so as not to reuse them again at a later stage.

From these three use cases we can derive a number of requirements for the entities modeling the Track-Hits association:

- It must be possible to create and make persistent associations between DataObjects.
- A client must be able to navigate from a DataObject in the TDS to a DataObject in persistent form transparently.
- It must be possible to create bi-directional associations, even between DataObjects created at different stages of the reconstruction chain.
- It must be possible to add information to the association and store it along.

---

<sup>9</sup> The hit object is created earlier in the reconstruction sequence, hence we speak of navigating *back*.



Clearly the first two requirements and the first use case they derive from can be satisfied using a `DataHandle`. A `DataHandle<HitCollection>` can for example be used to implement the Track to Hits unidirectional association mentioned in the first use case.

A more general solution that would cover the two remaining use cases as well could be to use *associative objects*. For example, an instance of a `TrackAssociatedHit` would contain a pair of `DataHandles` referring to a `Track` and a `Hit` associated to it. The `TrackAssociatedHit` object could also have association-specific information (e.g. track residuals). Each `TrackAssociatedHit` would be mapped into an “associatedHits” collection, indexed using the hit as key<sup>10</sup>. Forward navigation could be performed following the links from the `Track` object to the `TrackAssociatedHit` and from there to the `Hit` object<sup>11</sup>.

More interestingly, this scheme would allow creating and navigating back references without violating reconstruction “causality”. When a new `Track` is created, a new `TrackAssociatedHit` instance is created for each of the associated hits and added to the `associatedHits` collection. Note that the hit object itself remains untouched. When, given a hit, we want to find the “parent” track, we look up the relevant `TrackAssociatedHit` instance in the `associatedHits` collection and use it to navigate to the track.

## **F) Data Proxies**

According to the Gang of Four Pattern book<sup>12</sup>, the intent of a Proxy is to:  
*Provide a surrogate or placeholder for another object to control access.*  
*One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.*

As we mentioned above we propose to use `DataProxies` to represent `DataObjects` known to the `StoreGate`. The `DataProxy` will take care of the creation of the `DataObject` it represents and could also control the access to the `DataObject`.

### **F.1 Access Control**

`DataProxies` could be used to implement a complex `DataObject`-based access control policy. For example, to allow only certain modules write into an existing `DataObject`, or to allow only certain users (or groups, or batch classes) to mount a tape to read a `DataObject`. Since the `DataProxy` “intercepts” the `DataObject` interface and present it to the `Knowledge` object, neither the `DataObject` nor the `Knowledge` object need to know about the details of the access control policy.

---

<sup>10</sup> A likely implementation would be

```
std::multimap< DataHandle<Hit>, DataHandle<TrackAssociatedHit> >
```

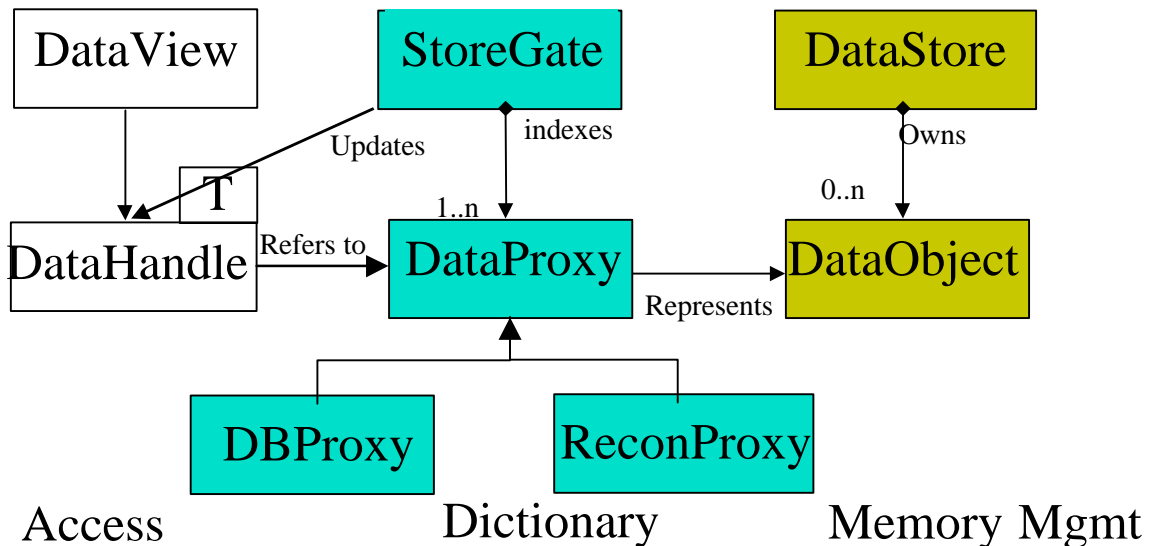
<sup>11</sup> Of course, forward navigation would often be implemented in a simpler and more efficient way using the `DataHandle<HitCollection>` `Track` data member mentioned above.

<sup>12</sup> Design Patterns, Gamma, Helm, Johnson & Vlissides, An Addison Wesley publications.



## F.2 Data Object Creation

When a data object is requested via the StoreGate, the returned DataHandle refers to one of the DataProxies known to the StoreGate<sup>13</sup>. When the DataHandle is actually used by the client, the proxy will return a reference to the requested DataObject if this is already in the TDS. If the requested DataObject is not there, it will create the new DataObject instance, register it with the TDS and return a reference to the client. During the StoreGate initialization, one or more Proxy classes are associated with a given DataObject type. For example, a generic ObjDBProxy<LArCellCollection> class could read cell collections using Objectivity; a LArDetDBProxy would take care of the LAr detector description constants; an ObjDBProxy<LArClusterCollection> could provide the reference set of reconstructed clusters from tape; while a LArClusterRecoProxy could be used to reconstruct on demand a new set of clusters.



How does a Proxy based scheme compare with the existing Gaudi conversion scheme? In the current scheme, the OpaqueAddress has a role similar to the DataProxy, in that it contains all the information needed to create the DataObject it refers to. The actual creation, on the other hand is the result of a rather complex interaction among the Data Service, the Persistency Service, the Conversion Service and the Converter. A DataProxy can be seen as an OpaqueAddress that knows how to create the object it refers to, and interacts as needed with the Services and the Converters.

## F.3 Reduced Coupling

The third, may be less obvious, but possibly most important, advantage of a DataProxy is that it further reduces the coupling among Data and Knowledge objects. We mentioned in

<sup>13</sup> If the StoreGate dictionary can not associate a proxy to the client retrieve, the system has been badly configured and an exception should be thrown.



the motivation section how the separation of Data and Knowledge objects insulate the former from the fast evolution of the latter. Introducing DataProxies, we also remove any dependency of the Knowledge objects from the DataObjects implementation<sup>14</sup>. If a MuonFinder algorithm needs a track collection as input, you could transparently use a five-year old track collection on tape, or the latest and greatest version of the track collection class that has been reconstructed on the fly<sup>15</sup>. Switching from one to the other will be just a matter of changing the MuonFinder configuration in the job options.

## Summary

It must be emphasized that much of the discussions on the requirements of the Event Model, the strengths and weaknesses of the myriad of available possibilities are still ongoing. We have thus far benefited tremendously from the experience gained by other experiments, particularly BaBar, CDF, CLEO, DØ and LHCb. It is clearly understood that a robust design is necessary that not only satisfies that large range of client requirements but is also simple and elegant to use. We have developed a prototype model to test some of the described features while maintaining the Gaudi backbone infrastructure. The main goal of the prototype was to put in place a stable client interface with minimal set of functionality's early on and be in a position to extend its behavior while maintaining the important client interface. Some of the functionality's that have been implemented in the prototype are:

- The ability to store and access DataObjects by types.
- Storing and retrieving data objects with Keys.
- Returning a DataHandle (a const-pointer) to the DataObject in the TDS.
- Ability to use simple selectors that is able to filter on collections.
- Ability to browse the TDS of all collections of a given type.

The prototype has been tested using the Gaudi Tutorial examples and is in the process of being made public to gain further exposure and testing from a wider set of software developers. The feedback will in no doubt strengthen the overall design of the Event Model and help in identifying the optimal solution that satisfies the ATLAS requirements.

## Acknowledgements

The editors would like to thank all members of the EDM working group, and the other ATLAS collaborators who contributed to the discussion through mailing lists, private conversations, and such. We are particularly grateful to many colleagues from other experiments who shared their experiences with us, in particular M. Frank and P. Mato from LHCb, M. Paterno and S. Snyder from DØ, S. Patton from BaBar.

---

<sup>14</sup> And, of course, from the specifics of the object creation mechanism.

<sup>15</sup> Even if data object interfaces tend to be stable, the Track interface may have changed in five years. In this case the DataProxy would rather be an Adapter, a related pattern which provides a different interface to the object it adapts.