LHCb

GAUDI

LHCb Data Processing Applications Framework

Architecture Design Document

Reference:	LHCb 98-064 COMP
Version:	1.0
Created:	November 9, 1998
Last modified:	November 24, 1998
Prepared by:	LHCb software architecture group
	Editor: P. Mato

This document has been prepared with Release 5.5 of the Adobe FrameMaker® Technical Publishing System using the User's Guide template prepared by Mario Ruggier of the Information and Programming Techniques Group at CERN. Only widely available fonts have been used, with the principal ones being:

Running text:	Times New Roman 10.5 pt on 13.5 pt line spacing
Chapter numbers and titles:	Arial 28 pt
Section headings	Arial 20 pt
Subsection and subsubsection headings:	Arial Bold 12 and 10 pt
Captions:	Arial 9 pt
Listings:	Courier Bold 9 pt

Use of any trademark in this document is not intended in any way to infringe on the rights of the trademark holder.

Contents

1

Introduction

1.1 Purpose of the document

This document is the result of the architecture design phase for the LHCb event data processing applications project. The architecture of the LHCb software system includes its logical and physical structure which has been forged by all the strategic and tactical decisions applied during development. The strategic decisions should be made explicitly with the considerations for the trade-off of each alternative.

The other purpose of this document is that it serves as the main material for the scheduled architecture review that will take place in the next weeks. The architecture review will allow us to identify what are the weaknesses or strengths of the proposed architecture as well as we hope to obtain a list of suggested changes to improve it. All that well before the system is being realized in code. It is in our interest to identify the possible problems at the architecture design phase of the software project before much of the software is implemented. Strategic decisions must be cross checked carefully with the experts (the reviewers) because they are essential for the system which is being developed, specially if the system needs to be operational for very long time scales.

1.2 Scope of the system

The goal of the project is to build a framework which can be applied to a wide range of physics data processing applications for the LHCb experiment. We would like to cover all stages of the physics data processing: physics and detector simulation, high level software triggers, reconstruction program, physics analysis programs, visualization, etc. An also to cover a wide range of different environments such as interactive non interactive applications, off-line and on-line programs, etc.

Using a framework, the development of the LHCb software system is made easy because major functional elements can be reused. The framework is the implementation of the architecture, thus the importance of developing an architecture which can “work” under the various environments and data processing stages. So, it is important that the framework has sufficient knobs, slots and tabs that can be adapted to current problem and integrated to other frameworks.

1.3 Definitions, acronyms and abbreviations

1.3.1 Definitions

Architecture The software architecture of a program or computing system is the structure or structures of the system, which comprises software components, the externally visible properties of those components, and the relationships among them.

Framework A framework represents a collection of classes that provide a set of services for a particular domain; a framework exports a number of individual classes and mechanisms that clients can use or adapt. A framework realizes an architecture.

Component A software component is a re-usable piece of software that has a well specified public interface and it implements a a limited functionality. Sofware components achieve reuse by following standard conventions.

1.3.2 Acronyms

CERN European Organization for Nuclear Research

URD User Requirements Document

ADD Architecture Design Docuument

UML Unified Modeling Language

ODBMS Object-oriented Data Base Management System

1.4 References

[1] P. Binko ed., “LHCb Computing Tasks”, LHCb/98-042 COMP

[2] P. Maley ed., “LHCb Application Framework: URD document”, LHCb/98-XXX COMP

[3] L. Bass et al., “Software Architecture in Practice”, Addison-Wesley, 1998

[4] G, Booch, “Object Solutions: Managing the object-oriented projet”, Addison-Wesley, 1996

[5] L. Lakos, “Large-scale C++ software design”, Addison-Wesley, 1998

1.5 Overview of the document

The first chapter of the document is the introduction containing the purpose, scope and series of definition of terms. Chapter 2 includes the overview of the system together with the description of its basic functionality.The overall system design is described in chapter 3. This system design includes the first level decomposition of the system into hardware and functional components with diagrams. Starting from chapter 4, each chapter is devoted to the description of a single component. It includes what the component is, what is the purpose, what it does, how it is decomposed, the interfaces and the dependencies with other components.

2

System Overview

In this chapter we introduce the LHCb event data processing problem and the scope of the solution we are proposing. For more details of what the problem is in terms of variety and scale please refer to the *Technical Proposal* supporting document [1] and the collection of requirements and scenarios collected in the document [2].

2.1 Computing tasks

The software system we are designing covers all software tasks needed for processing the event data. It spans the domains of on-line and off-line computing. It includes the algorithms to filter interesting events from background (so called high level triggers), as well as the full reconstruction and analysis tasks. The overview of these tasks are shown in Illustration 2.1 using a dataflow diagram.

As we can see, the event data processing system will consists of a series of processes or tasks that will transform the data collected from the detector into physics results. The data processing is done in various stages following the traditional steps: data collection and filtering, physics and detector simulation, reconstruction and finally physics analysis. The development of all these data processing tasks will involve in one hand computing specialists to build the framework and the basic components and on the other hand the people specialized on each sub-detector that will build the specific code which will be needed for the reconstruction, simulation of each of the sub-detectors and its final combination. In addition, there will be the people developing the data analysis programs to produce the final physics results.

2.2 System Context

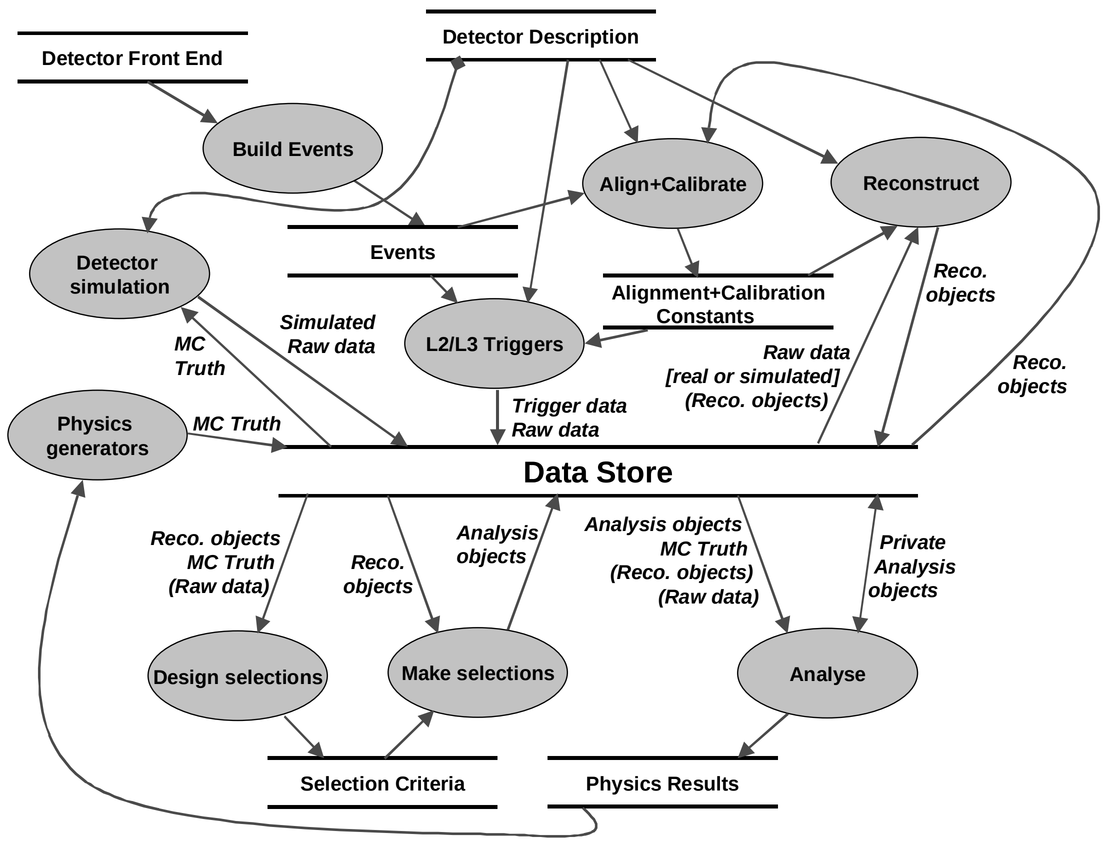
The software framework will play a central role. Practically all members of the colaboration will interact with it in one way or another. It will provide the skeleton for the reconstruction and analysis of data from the experiment. It will be used for producing simulated data. Monitoring of the experiment during data taking may also make use of the facilities supplied by the framework.

Illustration 2.1 shows the relationship between the framework, the experiment and people.

We have categorized the people who work with the framework into four groups. This categorization is not intended to be exclusive, it is a categorization of interaction rather than of people and many people will belong to several groups

1. Physicists. These people are principally interested in getting results. Any software they produce is for private use only. They are interested in analysing reconstructed data. They produce histograms, statistical distributions which they fit to extract parameters, etc.. If they input anything into the system it is in terms of ideas and physics processes (e.g. for event generators).
2. Physicist developers. These people will contribute to a big fraction of the system code in terms of number of lines. Their principal occupation is to implement components within the provided framework. These components are such things as: Detector simulation and response code; reconstruction code; etc.. We differentiate between physicists and physicist developers, because individuals fullfilling the later role are supplying something that will be used by many other people. Thus the approach to producing software must be that much more disciplined than for producing private code. In addition this activity will require more knowledge of the framework than that required by the average physicist user.
3. Configuration managers. These people are responsible for the management of data production (Monte Carlo, Reconstruction,...), versioning of detector geometry, calibration, alignment, etc. etc.. They do not necessarily need to write much code, but they will probably need to be conversant with database management tools.
4. Framework developers and maintainers. These people are responsible for the design, implementation and maintenance of the framework itself.

(a)



(b)

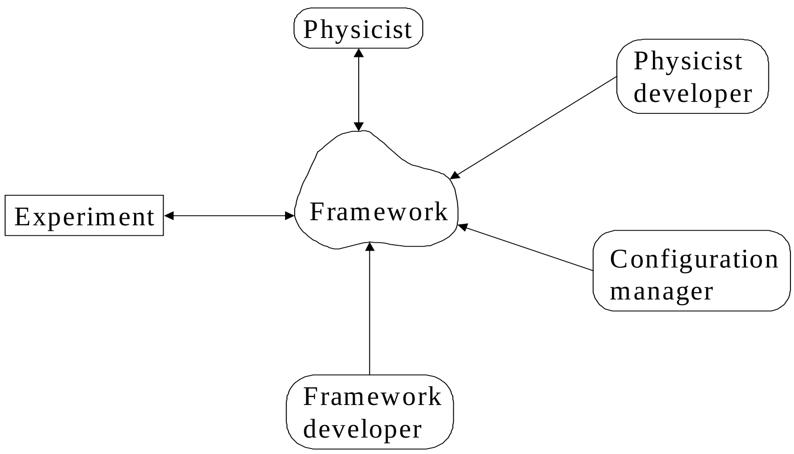


Illustration 2.1 (a) Dataflow diagram showing all the main tasks and data flows for the LHCb computing
(b) Context diagram for the framework.

2.3 Project Strategy

The ultimate aim of the project is to provide a software framework useable by the entire LHCb collaboration for the simulation, reconstruction and analysis of p-p interactions at the LHC.

The first step in achieving this goal is of course to understand the requirements of the future system users. This is not as simple as it might seem as it is very difficult to start from what is essentially a blank piece of paper and write a list of formal requirements.

The design of the framework is driven by the requirements of the physicists who will develop reconstruction and simulation code and who will use the framework to do analysis of the data. These people know in broad terms what they want to do, but the details are as yet unknown. The development of this type of software is a very exploratory process: ideas which work are kept, those which don't are thrown away. Thus we know that the system must be flexible, but in order to be able to implement something we must establish the boundaries of this flexibility.

Our approach, then, to establishing the system requirements has been a combination of using our own past experience (as most of us have a background in HEP) along with interviewing a subset of the future users (physicist developers primarily). The aim of these (very informal) interviews was to try to step through the likely processes of simulation, reconstruction and analysis and to extract example usages of the system. Additionally, the experiences that people have had with software on other experiments is also a valuable source of information. For example, a physicist may remember trying to do an analysis which required a lot of hacking, or contorted coding because that particular usage had not been forseen by the software designers. This kind of example is very important to us while considering the current project.

Once this first step of identifying the main system requirements is in a healthy condition (it probably never is finished) we can start on the design and implementation. However it is crucial that the development of the software does not proceed in isolation but intimately involves the future users. Otherwise there is a very large risk of producing a framework which just does not match with their wishes. Additionally, time does not stop while the software is developed: physicists will continue to work, to develop algorithms and make studies. If they do not have a framework provided for them, they will go and find one themselves. This inevitably leads to a fragmentation of the collaboration with regards computing and a needless duplication of effort as, for example, many people develop their own visualisation software.

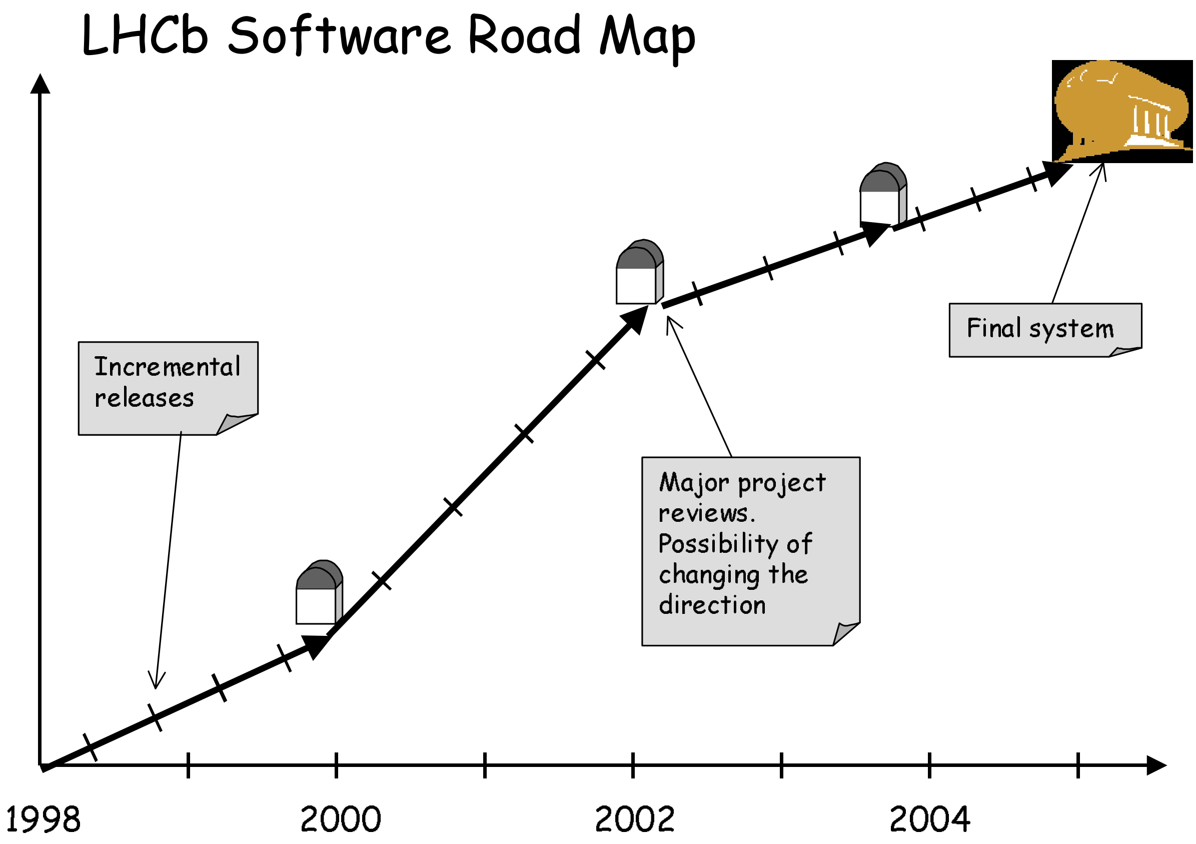
In order to avoid this we will approach the final software framework via incremental steps. Each step adding to the functionality of the framework. The only way to do this without producing an amorphous blob of unmaintainable software is to use an architecture driven approach. That is, to identify a series of large scale components with definite functionality and well defined interfaces which interact with each other to supply the whole functionality of the framework. All of these components are essentially decoupled from each other.

Once the architecture is relatively stable, implementation can begin. Components can be implemented one at a time and in a minimal manner, i.e. supplying sufficient functionality to do their job, but without the many refinements that can be added at a later date. In addition components can be replaced easily by anything which implements the appropriate interface and provides the appropriate functionality. Thus we can make use of “third-party” software.

Thus our strategy for arriving at the final project goal with the rest of the collaboration in-tow is to go through many short cycles of incremental implementation and release. At each stage feedback from the users will make sure that we do not stray from what they want and also set the priorities for what the following release should contain.

Each increment would first be tested by selected users.

The first release will consist of sufficient software to allow users to read simulated events which already exist in the form of ZEBRA files.


LHCb Software Road Map
Incremental releases
Major project reviews. Possibility of changing the direction
Final system
1998
2000
2002
2004

3

System Design

The architecture of the system is described in terms of the components we have identified and their interactions. A software component is a part of the system which performs a single function and has a well defined interface. Components interact with other components through their interfaces.

The notation we are using in this document to specify the architecture is the Unified Modeling Language (UML). This notation is not completely adequate for describing architectures in general, but in our case it seems to be sufficient and it has the advantage that is widely known, thus we do not need to introduce other notations. We are going to use mainly *object diagrams* to describe a snapshot of the components and their relationships at a given moment in time, *class diagrams* to show the software structure and *sequence diagrams* to describe some of the use cases or scenarios.

3.1 Major design criteria

Before we start with the description of the architecture we have crafted we need to explicitly document what have been our design criteria and what are our strategic decisions.

Clear separation between “data” and “algorithms”

Despite our intention to produce an object oriented design, we have decided to separate data from algorithms. For example, we are thinking to have “hits” and “tracks” as basically *data objects* and to have the *algorithms* that manipulate these data objects encapsulated in different objects such as “track_fitter” or “cluster_finder”. The methods in the *data objects* will be limited to manipulations of internal data members. An *algorithm* will, in general, process *data objects* of some type and produce new *data objects* of a different type. For example, the cluster finder algorithm, produces cluster objects from raw data objects.

Three basic categories of data: event, detector and statistical data

We envisage three major categories of *data objects*. There will be the *event data* which is the data obtained from particle collisions and its subsequent refinements (raw data, reconstructed data, analysis data, etc.). Then, there will be *detector data* which is all the data needed to describe and qualify the detecting apparatus in order to interpret the *event data* (structure, geometry, calibration, alignment, environmental parameters, etc.). And finally, we will have *statistical data* which will be the result of some processing applied to a set of events (histograms, n-tuples, etc.).

Clear separation between “persistent data” and “transient data”

A main feature of our design is that we separate the persistent data from the transient data for all types of data e.g. event, detector description, histograms, etc. We think that physics algorithms should not use directly the data objects in the persistency store but instead use pure transient objects. Moreover neither type of object should know about the other. There are several reasons for that choice:

- The majority of the physics related code will be independent of the technology we use for object persistency. In fact, we have foreseen to change from the current technology (Zebra) to an ODBMS technology preserving as much as possible the investment in terms of new developed C++ code.
- The optimization criteria for persistent and transient storage are very different. In the persistent world you want to optimize I/O performance, data size, avoid data duplication to avoid inconsistencies, etc. On the other hand, in the transient world you want to optimize execution performance, ease of use, etc. Additionally you can afford data duplication if that helps in the performance and ease of use.
- To plug existing external components into our architecture we will have to interface them to our data. If we interface them to our transient data model, then the investment can be reused in many different types of applications requiring or not requiring persistency. In particular, the transient data can be used as a bridge between two independent components.

Data centered architectural style

The architecture we are envisaging should allow the development of *physics algorithms* in a fairly independent way. Since many developers will be collaborating in the experiment software effort, the coupling between independent algorithms should be minimized. We have envisaged using a *transient data storage* as a means of communication between algorithms. Some algorithms will be “producing” new data objects in the data store whereas others will be “consuming” them. In order for this to work, the newly produced data objects need to be “registered” somehow into the data store such that the other algorithms may have the possibility of identifying them by some “logical” addressing schema.

Encapsulated “User code” localized in few specific places: “Algorithms” and “Converters”

We need to take into account the need to customize the framework when it is used by different event data processing applications in various environments. Most of the time this customization of the framework will be in terms of new specific code and new data structures. We need therefore to create a number of “place holders” where the physics and sub-detector specific code will be later added. We are considering two main places: *Algorithms* and *Converters*.

All components with well defined “interfaces” and as “generic” as possible

Each component of the architecture will implement a number of interfaces (pure abstract classes in C++) used for interacting with the other components. Each interface consists of a set of functions which are specialized for some type of interaction. The intention is to define these interfaces in a way as generic as possible. That is, they should be independent of the actual implementation of the components and also of the concrete data types that will be added by the users when customizing the framework.

Re-use standard components wherever possible

Our intention is to have one single team with an overview of the complete LHCb software system covering the traditional domains of off-line and on-line computing. We hope in this way to avoid unnecessary duplication by identifying components in the different parts of the system which are the same or very similar. We intend to re-use standard and existing components wherever possible.

Integration technology standards

We are not currently in a position to select a standard integration technology that provides the glue between the different components of the architecture. However, we are aware of the importance of selecting a technology and standardizing on it to guarantee a smooth integration and facilitate re-use of commercial components. For this reason, we have tried to follow some of the ideas behind the more popular integration technologies (CORBA, DCOM, JavaBeans) such that their later adoption will not be traumatic.

3.2 Object diagram

We introduce our description of the architecture by an *object diagram* showing the main components of system (Illustration 3.1). We know that object diagrams are not the best way to show the structure of the software but they are very illustrative for explaining how the system is decomposed. They represent a hypothetical snapshot of the state of the system, showing the objects (in our case component instances) and their relationships in terms of navigability and usage.

Algorithms and Application Manager

The essence of the event data processing applications are the physics algorithms. We encapsulate these into a set of components that we called *algorithms.* These components implement a standard set of generic interfaces. *Algorithms* can be called without knowing what they really do. In fact, a complex *algorithm* can be implemented by using a set of simpler ones. At the top of the hierarchy of *algorithms* sits the *application manager.* The *application manager* is the “chef d’orchestre”, it decides what algorithms to create and when to call them.

Transient data stores

The data objects needed by the *algorithms* are organized in various *transient data stores*. We have distributed the data over three stores as shown in the diagram. This distribution is based on the nature of the data itself and its lifetime. The event data which is only valid during the time it takes to process one event is organized in the *transient event store*. The detector data which includes the detector description, geometry, calibration, etc. and generally has a lifetime of many events is stored in the *transient detector store*. Finally, the statistical data consisting of histograms and n-tuples which generally have a lifetime of the complete job is stored in the *transient histogram store*. It is understood that the three stores behave slightly differently, at least with respect to the data lifetime (the event data store is cleared for each event), but their implementations have many things in common. They could be simply different instances of a common component.

Services

We have defined a number of components which should offer all the services directly or indirectly needed by the *algorithms*. The idea here is that we would like to offer high level services to the physicist, so that they can concentrate on developing the physics content of the algorithms and not on the technicalities needed for making everything work. We call this category of components *services*.

Some examples of services can be seen in the object diagram. For instance, there are services for managing the different transient stores (*event data service*, *detector data service*,...). These services should offer simplified data access to the algorithms. Another class of service are the different *persistency services*. They provide the functionality needed to populate the transient data stores from persistent data and vice versa. These services require the help of specific *converters* which know how to convert a specific data object from its persistent representation into its transient one or the other way around. Other services like the *job options service*, *message service*, *algorithm factory*, etc. which are also shown in the diagram offer the service which its name indicates. They will be described in more detail later in the document.

Selectors

We have envisaged a number of components whose function will be to select. For instance, the *event selector* provides functionality to the *application manager* for selecting the events that the application will process. Other types of selectors will permit choosing what objects in the transient store are to be used by an algorithm, by a service, etc.

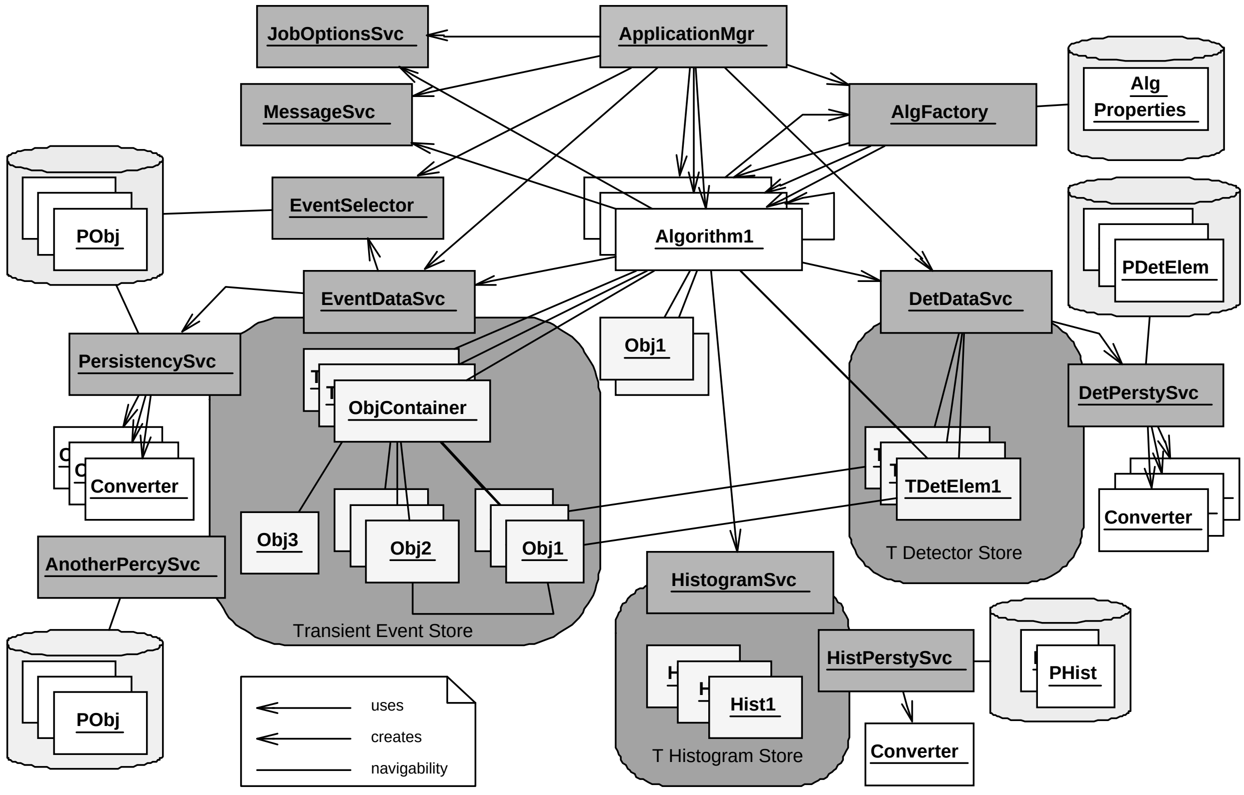


Illustration 3.1 Object diagram of the proposed architecture covering the kernel of any data processing application

3.3 Classification of classes and component model

Having introduced the decomposition of the system in the previous section, it is important now to show the structure of the software in terms of what type of classes we are envisaging and their hierarchies. Table 3.1 lists the main classes which will form the kernel of the architecture

.

Table 3.1 Classification of classes

Application Managers	One per application. The “chef d’orchestre”.
Services	Offering specific services with well-defined interfaces. Different concrete implementations depending of specific functionality.
Algorithms	Physics code. Nested algorithms. Simple and well-defined interface.
Converters	In charge of converting specific event or detector data into other representations.
Selectors	Components to process a selection criteria for events, parts of events or detector data
Event/Detector data	The data types that the algorithms and converters use. Simple behavior.
Utility classes	All sorts of utility classes (math & others) to help with the implementation of the algorithms.

The class hierarchies for the services and algorithms are shown in Illustration 3.2. These are not meant to be complete and in their final form. We can see that all services will inherit from a common service which implements a generic *IService* interface. This interface provides the common functionality which is needed for each service. For example, it provides a reference counting mechanism such that, any service that is no longer referenced will automatically be deleted from the system. It also provides some identification of the service that will be needed to locate the service during the initialization phase of the algorithms and other components.

In general, any component can implement more than one interface[1]. By doing that, we intend to specialize each interface to a set of highly related functions which will be used typically by one type of client. For example, the *Algorithm* component implements the *IAlgorithm* interface which is used by the *Application Manager* or other *Algorithms* to process physics events. However it also implements the *IProperty* interface which allows the application Manager or any interactive component to change the behavior of the algorithm by changing some of its internal properties. Clients of any component will have one or more references to the interfaces of the component but never to the concrete component object. This is essential to ensure minimal coupling between components. In addition, all the interfaces (pure abstract classes in C++) will inherit from a common ancestor called *IInterface*. This will allow us to provide a mechanism for querying an interface of a component for a reference to any other interface[2].

1 We follow the idea of Java, having single concrete class inheritance and implementation of multiple interfaces.
2 This mechanism is very similar to the one provided by the COM component model of Microsoft.

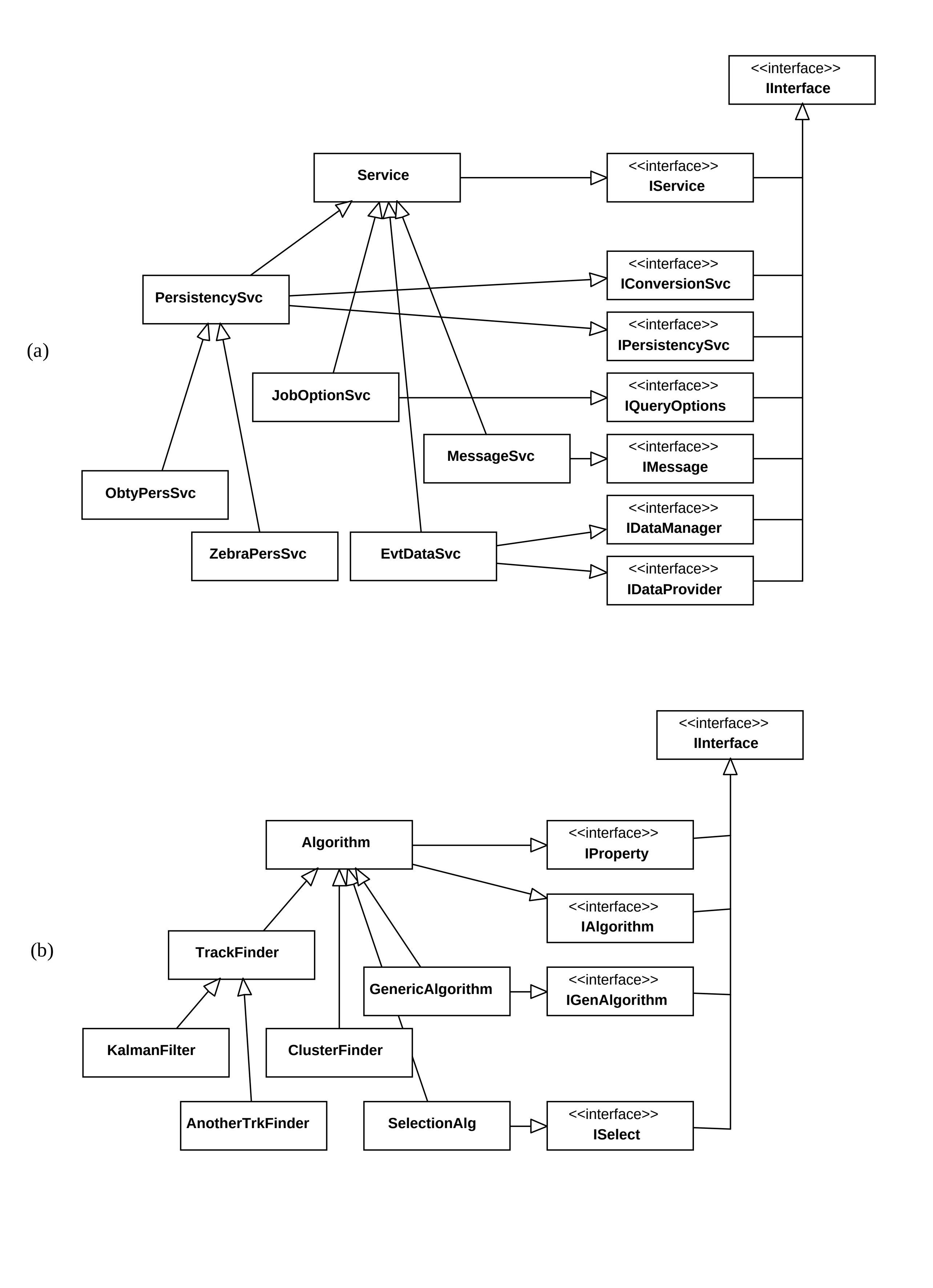


Illustration 3.2 Class hierarchies for some of the types of classes in the architecture: (a) services, (b) algorithms

3.4 Transient Data Model

The organization of the data in the *transient data store* will be a tree of data objects. This structure resembles very closely a typical file system with files and directory files, where directory files may contain also some data attributes. This is shown in Illustration 3.3 (a). Each data object can potentially be a node of the tree and in addition also contain its own data members or properties. For example, the *Event* object is the root node for all the event data and it has a set of properties, e.g. event number, event time, event type, etc.

Any object in the data store needs to be uniquely identified. As in the case of the file system, the identification (i.e. file name) is unique at the level of its container. The “full path” identification that uniquely identifies the object in the store is then made by concatenating the identifiers (names) of all the ancestor nodes with its own identifier.

We are aware that most of the event data objects will be very tiny data objects. For example, for each event we will have hundreds if not thousands of hit objects each of which will be a few bytes long. So, we do not want these tiny objects to incur a big overhead when being managed in the transient data store (the same arguments applies for the persistent store). Therefore, we foresee objects in the store that will be normal identifiable objects and in addition be containers of small objects. These small objects themselves are not identifiable directly, but rather by containment.

To summarize, we envisage to have the following types of objects in the transient data store:

- Identifiable objects with data members (properties) which are also nodes from where other objects hang.
- Normal identifiable objects
- Simple objects which are contained in one identifiable object.

Following the analogy with the file system it is equivalent to say that what we have in the data store are “directories with properties”, “files” and “records”. It is clear that a directory is a special type of “file”.

Having this strong hierarchical structure between data objects (aggregation) does not preclude other kinds of relationships between the different objects. For example, as we can see in the Illustration 3.3 (b), we can have a hierarchy consisting of the event root, raw event, a number of sets containing hits and a number of sets containing tracks. On top of this hierarchy we can have a relationship between hits and tracks.

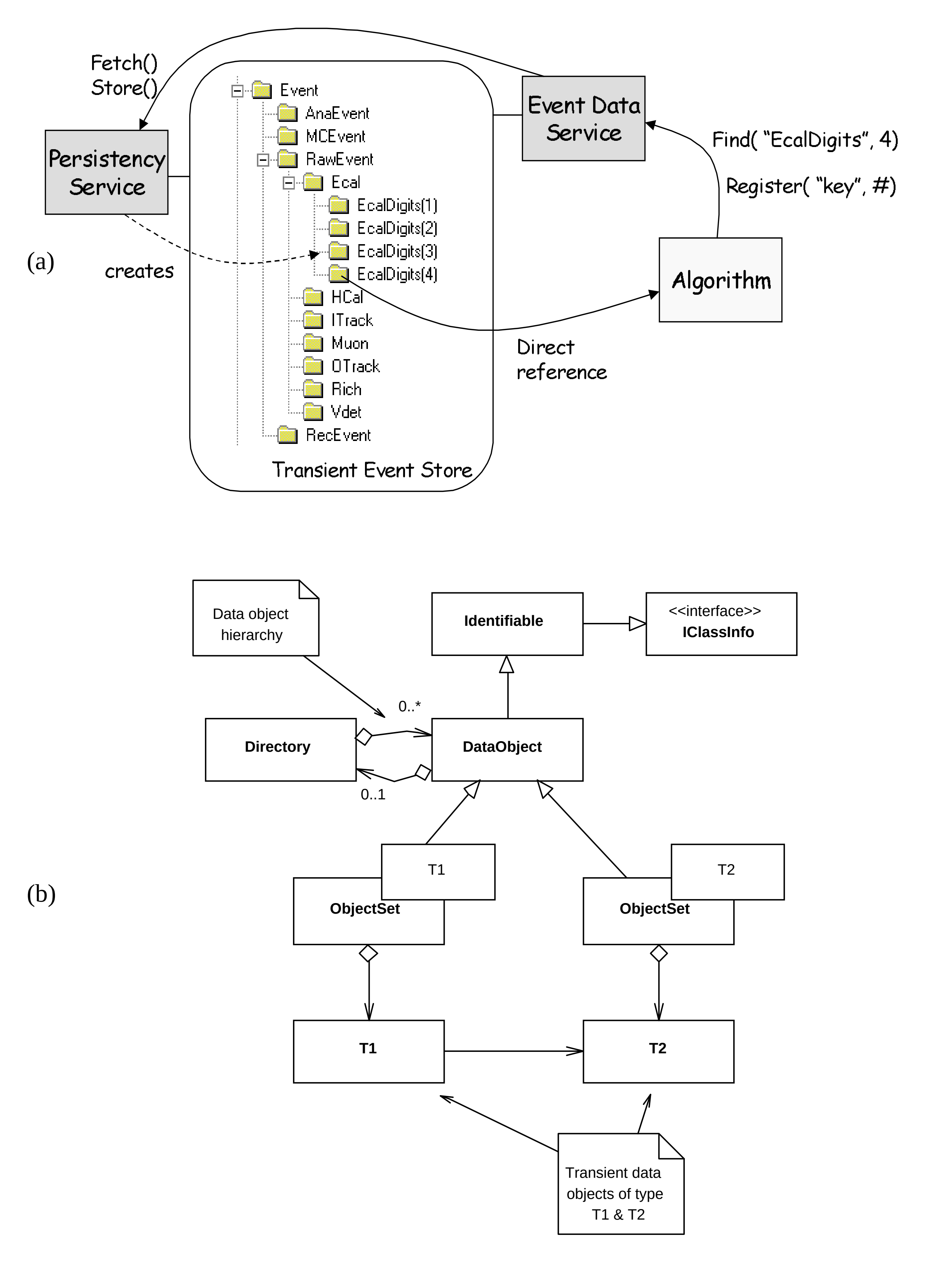


Illustration 3.3 Structure of the transient data store (a). Class diagram for the transient data model (b).

3.5 Algorithms and the Transient Data Store

The *Algorithm* component is the support structure for real computational code. Any code that a user would like to be executed within the framework must conform to the *Algorithm* component specification. This allows us to execute *Algorithms* without knowing what is its actual implementation.

Complex *Algorithms* will be implemented by executing one or more basic ones. These *Algorithms* that combined produce a high level algorithm need to exchange some data to perform its function. The way this data communication is done is by using the transient data store. As shown in Illustration 3.4, same algorithms are putting data objects in the store and others are retrieving them creating the illusion of data being sent from one algorithm to the other.

3.6 Transient and Persistent data representations

As mentioned in the architecture design criteria, we have chosen to have different object representations for the transient and for the persistent store as opposed to having a single representation of “persistent capable” objects. In that way we hope to be able optimize both worlds independently and decouple from the persistent storage technology.

The problem that now faces up is the need of converting objects from one representation to the other and vice versa. To solve this problem there are several options, one of them is to describe the user data types within the framework (metadata) and have utilities that using this metadata are able of doing the conversion. This approach is elegant and relatively easy for basic data types. However it is extremely complicated when converting objects with arbitrary relationships and especially with relationships between the different data stores (event and detector). The other possibility is to code the conversion specifically for each data object type. For the time being this is the option we have taken. Where this code will be siting is another question that we should address. We want to put the conversion code neither in the transient class nor in the persistent class. So, we have created a new type of component called *Converter* that will have a common interface and will be called by the persistency service when an object needs to be converted. The tricky part is that we need to be able to identify the type of the data object we want to convert in order to call the appropriate *Converter.* This can be done by hard-wiring in the constructor of each data object a class identifier or by using run time type information.

Each *Converter* will be specific to the data type it is in charge of converting. In that context, the *Converter* can perform more complicated operations than just to convert one-to-one the persistent or the transient object. It can deal with the fact that maybe in persistent storage many tiny objects are combined into a single one (minimize overhead in space and I/O) and that when it is converted to the transient representation it is in fact expanded to the individual objects. This kind of flexible functionality is possible if the code is written specifically but it would not be possible if the metadata approach were used.

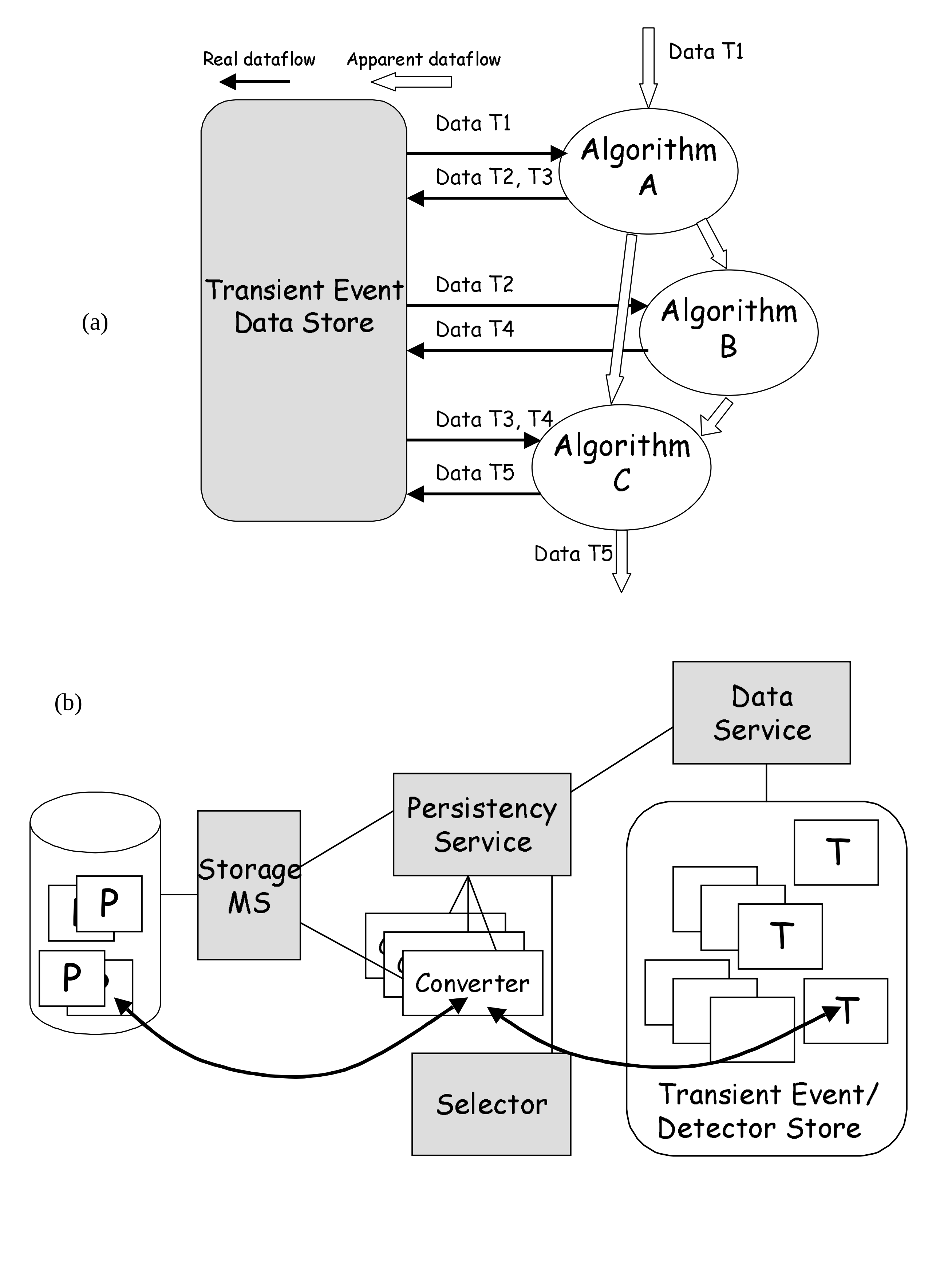


Illustration 3.4 (a) The interaction of Algorithms with the transient data store
(b) Persistent and transient representations

3.7 Links between event and detector data

We have organized the data objects needed by the *Algorithms* in three transient stores as has already been mentioned. The *transient detector store* contains the detector data which is needed to process the event data. For example, in the case of the reconstruction of *raw* data, we will find in the *transient detector store* the mapping between channel numbers and detector wire numbers or detector cell numbers. This information is essential for converting hits in electronics channels into hits in space. It is desirable that the transient representation of the set of *raw* digits belonging to a detector module has a reference to the object containing this mapping. This would improve the modularity of the Algorithms implementing the reconstruction. To do that, we need to have these references setup at the time the event data is converted into the transient representation (perhaps this relationship is not even exist explicitly in the persistent representation). The *Converters* are responsible of implementing these relationships.

The problem now is that we need to make sure that the correct (i.e. valid for the current event) detector data is referenced by the event data. This is especially important for slow control environmental parameters, alignment, calibration constants, etc. Typically these detector data objects should have a sort of validity range. A given set of alignment parameters is valid from a given moment in time to another time in the future. Knowing that a time stamp is always associated to each event (perhaps Monte Carlo events should have negative time stamps) the framework should make sure that the detector objects in the *transient detector store* are valid for that event if this is not the case the framework should make the necessary actions in order to update the detector object with the valid information. This synchronization should happen automatically without the *Algorithms* having to know about.

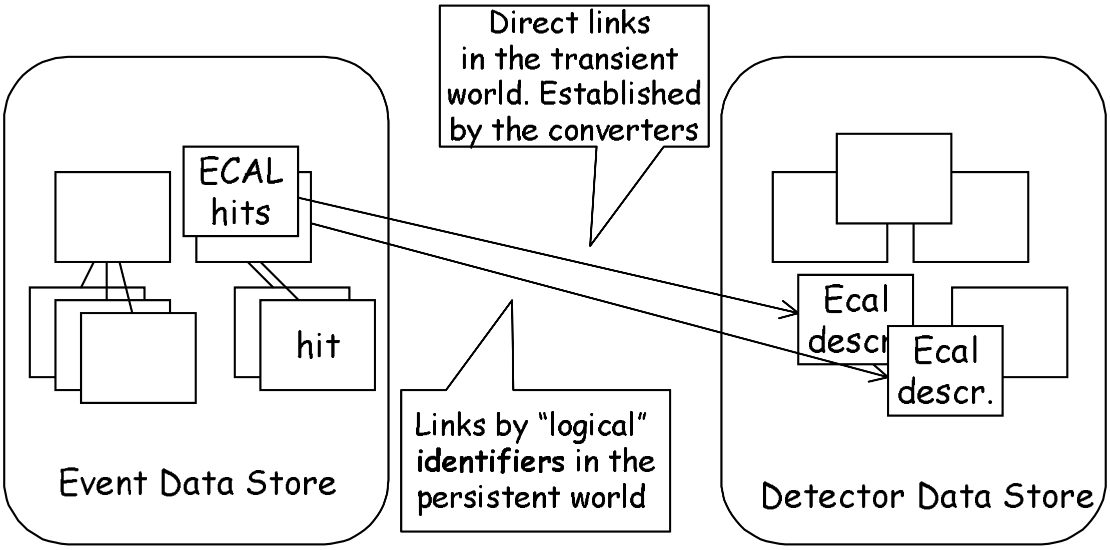
3.8 Data visualization

Another functionality required by the framework is the capability of visualizing *data objects* (event data, detector data, statistical data). For example, the event visualization consists of representing graphically in 3-D or in a projection a selected set of event data objects within a detector framework. The visualization is an aid for debugging the reconstruction algorithms, for monitoring in the on-line system, for helping in the physics analysis, etc.

The strategy we have taken is similar to the persistent/transient approach. We do not want to couple the *data object* definitions to a particular graphics technology. In others words, we do not want to declare and implement a method “draw()” in each *data object*. What we want is to have a service, the graphical representation service (*GraphRepSvc*), that is responsible for converting the *data objects* into their graphical representation. This graphical representation may depend in general on the type of object we are converting. We tend to represent differently the various “physics objects” (i.e. it is not the same to represent tracks than calorimeter hits or clusters). In general, the physicists are the ones that will decide what is the more convenient way to represent the physical objects. Therefore the pattern used is the same pattern as for the persistent/transient conversions. We will have a set of graphical representation *Converters* that will be called by the service when an object of the type that the *Converter* can handle needs to be converted (i.e. displayed).

The output of the conversion can either be directly the calls to the graphics package to produce the visualization effect or an intermediate representation independent of the final graphics packages. The second approach offers better portability to various platforms without re-writing the converters at the expense of using the minimum common denominator of the various graphics packages.

(a)


Direct links
in the transient
world. Established
by the converters
ECAL
hits
hit
Event Data Store
Links by "logical"
identifiers in the
persistent world
Ecal
descr
Ecal
descr.
Detector Data Store

3.9 Components

The following list of components have been identified during the design of the architecture. We define them very briefly with just enough detail to allow us to explain how they interact in order to provide the main required functionality of the framework. More detailed descriptions can be found in later chapters of the document.

Application Manager (ApplicationMgr)

The purpose of the application manager is to steer the data processing application. It is in charge of bootstraping the application by creating all the required services and processing components. It also implements the so called "event loop" by sequencing the initialization phase and controlling the execution of the algorithms for each of the events that must be processed.

Algorithm interface (IAlgorithm)

An algorithm component has a single responsibility - to perform a computation on input data and produce output data. The actual data types and computation performed will depend on the concrete algorithm. Algorithms in general will have properties or parameters which tune the computation.

Converter interface (IConverter)

A data *Converter* is responsible for translating data objects from one representation into another. Concrete examples are: converters creating transient objects representing parts of an event from the persistent representations, converters creating a textual representation of data objects for printing to the alphanumeric terminal, etc. Specific *Converters* will be to be needed for each data type that needs to be converted.

Job options service (JobOptionsSvc)

The purpose of the job options service is to supply the options for the current job to other components of the architecture. It is assumed that facilities allowing the user to edit the options and to save or retrieve sets of them for future use are supplied outside this service.

Event Selector (EventSelector)

The event selector will be used by the end user to select which events with given physics properties from all the available events will be processed by the application. The event selector is the component which knows what is the next event to be processed.

Data Object (DataObject)

Any transient data object (event, detector, statistical) used by the physics algorithms and capable of being stored in and/or retrieved from the transient store will inherit from a common base *DataObject* class. This allows us on the one hand to write interfaces to services in a general way and on the other to force a structure in the transient storage.

Transient Data Store

The transient data store is a passive component of the architecture where transient representations of data objects are stored temporarily during the execution of the algorithms. It is the "logical" place where these data objects are stored.

Event Data Service (EventDataSvc)

The event data service manages the transient event data store. This component provides the necessary functionality to allow the algorithms and the application manager to locate data objects in the transient store and to register new ones.

Event Persistency Service (EvtPersistencySvc)

The event persistency service delivers event data objects from a persistent store to the transient data store and vice versa. The persistency service collaborates with the event data service to provide data requested by the algorithms in case the data are not yet in the transient store. This service requires the use of specific converters which will allow it to convert the data objects into, and from, their persistent representation.

Detector Data Service (DetectorDataSvc)

The detector data service manages the transient detector data store. This component is very similar to the event data service with regards to the retrieval and registration of data objects. However it is additionally responsible for the synchronization of detector information to the current event.

Detector Persistency Service (DetPersistencySvc)

The detector persistency service delivers transient detector data objects from the persistent store. This service will need to have knowledge of the model showing/representing the way different versions of the detector description, calibration, alignment are stored.

Histogram Data Service (HistoDataSvc)

The main purpose of the histogram service is to store histograms or other statistical data in the transient histogram store and to retrieve them when necessary. This component is very similar to the event data service with regards the retrieval and registration of data objects.

Histogram Persistency Service (HistoPersistencySvc)

The histogram persistency service is able to load and save transient statistical data objects to and from the persistent storage.

User interface (UI)

The user interface is the component through which the end user interacts with most of the components of the system. In general, the components do have interfaces to allow clients to configure and control them. The user interface component is the bridge between the internal interface (C++) and an interface suitable for human interaction.

Message Service (MessageSvc)

The message service publishes messages originating from other software components. It is the only component of the architecture that is allowed to transmit messages to the outside world. The service filters messages according to their severity and dispatches them to different output destinations.

Data Item Selector (DataItemSelector)

A selection is a set of references to objects within a transient data store. This store could be either an event data store, a detector data store or a histogram data store. A selector is a generic component which creates a selection.

3.10 Component Interactions

The basic functionality of the framework can be described in terms of the interactions of the components. This is described in this section. For this we have used some typical examples of the basic functionality.

3.10.1Application initialization and basic event loop

- The main program for the application creates an instance of an *ApplicationMgr*[1]. The *ApplicationMgr* is then in charge of creating the required services. It will first create the *JobOptionsSvc* to know about its own environment and will start creating all services needed by default[2]. In particular, it will create the *MessageSvc, EventDataSvc, DetectorDataSvc,* etc.
- All the services created will need to be initialized before they can be used. They also have the opportunity to interrogate the *JobOptionsSvc* to find out about changes in their default behavior.
- The *ApplicationMgr* will continue by creating the top level *Algorithms* with the help of the *AlgorithmFactory.* The *AlgorithmFactory* creates the *Algorithms* with some default properties from persistent storage.
- The *Algorithms* will then be initialized. Part of their initialization will consist of interrogating again the *JobOptionsSvc* for properties being overwritten by the end-user. The *Algorithms* which require other *Algorithms* will be created and initialized at this moment.
- The *ApplicationMgr* will then setup the collection of events that will constitute the basis of its iteration (loop) and the various transient storages. The event collection will be obtained by the *EventSelector* upon the specification of the selection criteria. The *transient event store* will be cleared and the first event root declared. The *transient detector store* will also be cleared and populated with the correct version of the detector description, calibration, etc.
- All the *Algorithms* will be notified that a run is about to start in order to create the required statistical objects. This is achieved by cascading the notifications within the nested *Algorithms.*
- The top level *Algorithms* will be called to process the first event by the *ApplicationMgr.* Each *Algorithm* will then interact with the *EventDataSvc* to get a reference for the objects needed for its processing. Each *Algorithm* is in charge of invoking the execution of the event on the *Algorithms* it is controlling.
- The *EventDataSvc* will delegate the fetching of the requested object to the *EvtPersistencySvc* if the wanted piece of data is not yet in the transient data store.
- Once the event has been processed by the *Algorithms*, the *ApplicationMgr* instructs the *EvtPersistencySvc* to save into persistent storage a list of data objects. This list has been setup by the *ApplicationMgr* using a *DataItemSelector.*
- The *ApplicationMgr* retrieves then the next event to be processed from the *EventSelector.* It clears the *transient event store,* loads the new event root and signals the *DetectorDataSvc* that a new event has been loaded in case an update has to be done to the *transient detector store.*
- The execution of the job continues until the event collection is completed. Then, the *ApplicationMgr* informs the *Algorithms* that the job is about to finish to give then an opportunity to save the required statistical objects.

---

1 Various concrete implementations may exists of the ApplicationMgr specialized for different environments such as batch oriented or interactive application, or the type of processing such as the official reconstruction program, user analysis program, etc.

2 The service creation will require the help of a service factory in order to decouple the ApplicationMgr from the concrete services.

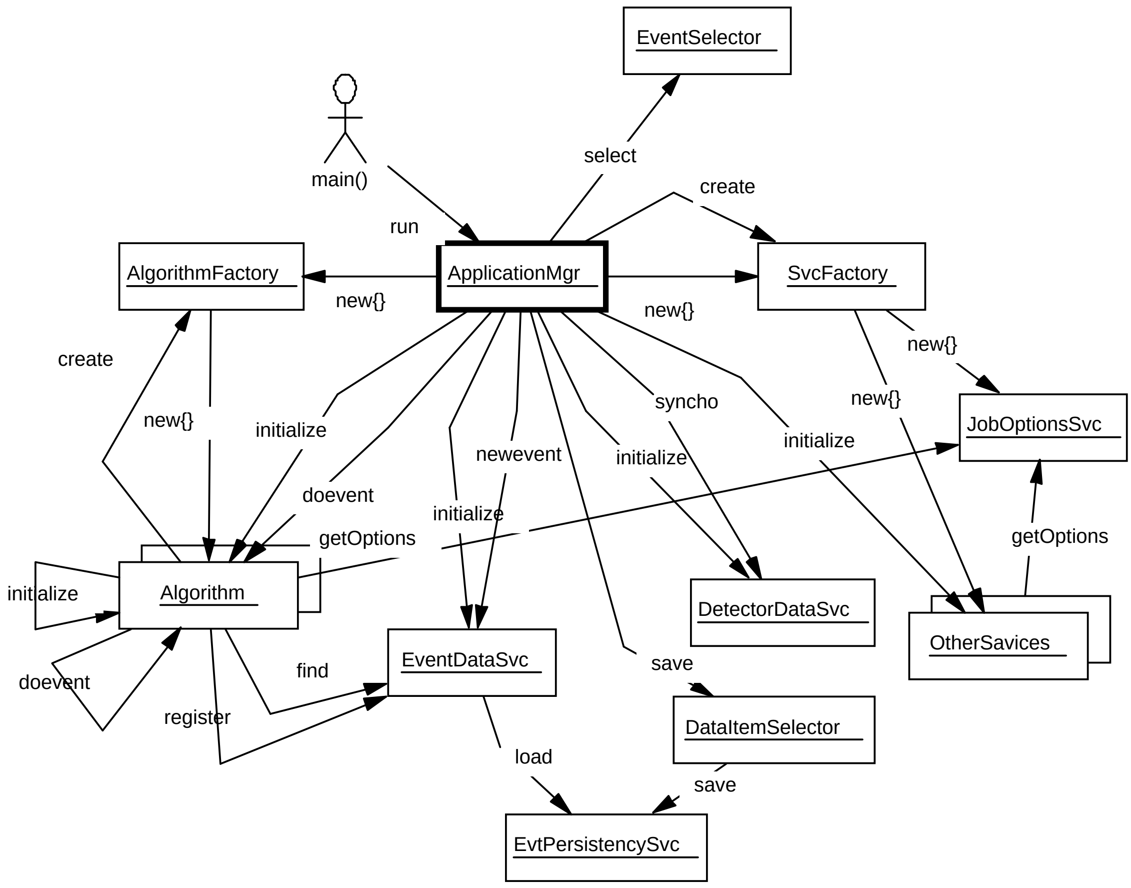


Illustration 3.6 Collaboration diagram for the application initialization and basic event loop

3.10.2Retrieving and storing event objects

The framework has to guarantee that we are able to retrieve and store data objects from the persistent storage:

- An *Algorithm* is asking the *EventDataSvc* for an object with a given identifier. If the object is not in the *transient event store*, i.e. it has not been registered previously, it will trigger an action on the *EvtPersistencySvc*.
- The *EventDataSvc* knows the “persistent address” of the required object because either it has the root from where the object is hanging or the address has been deduced from the identifier. It also knows the “type” of requested object from the root. Therefore, it uses this information to request to the *EvtPersistencySvc* to load the object.
- The *EvtPersistencySvc* selects the appropriate *Converter* from a set of *Converters* that have been declared at initialization by using the “type”. The *Converter* is called and locates the object in the persistent store and creates a transient representation of the same object initialized with the information of the persistent. It also fills the references of the new object to the other event or detector data objects using the *EventDataSvc* or *DetectorDataSvc*.
- The new created data object is then registered by the *EvtPersistencySvc* and makes it available to the *Algorithm* that requested it.
- Storing transient data objects after the processing is done following a similar collaboration between the *EvtPersistencySvc*, *EventDataSvc* and the adequate *Converter.*

3.10.3Detector data synchronization

The detector data must be kept synchronized with the event which is currently being processed. This is essential since the calibration, alignment and environmental data may change at any moment during the execution of the job.

- When a new event root is loaded in the *transient event store*, the *DetectorDataSvc* is informed by the *ApplicationMgr* together with the information of the event time.
- Each data object in the *transient detector store* has a validity time range. Assuming that the events are ordered in time (this simplifies the case), the *DetectorDataSvc* compares the current event time with the next time that any data becomes invalid.
- For most of the events, there is no action to be done since generally the validity ranges cover many events. But in some cases, the *DetectorDataSvc* will scan all the transient detector store and look for objects that need to be updated.
- The update of the object is done by the *DetPersistencySvc* which has the knowledge of how to find the new object in question knowing the event time.
- The object members are updated without the need to delete and create new objects. In this way references to them are still valid, so *Algorithms* can just use them without the need of knowing that a new calibration or alignment is being used.

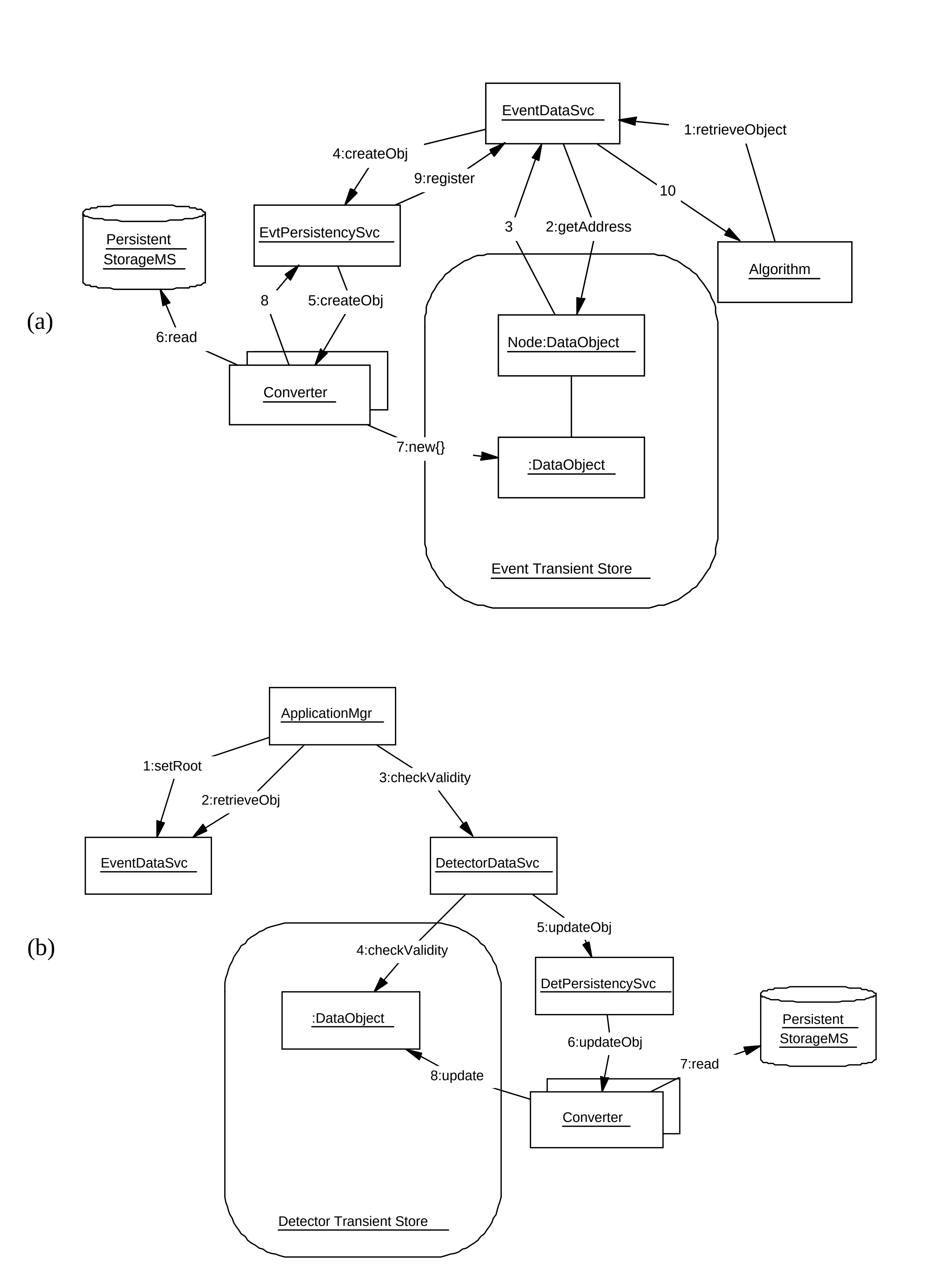


Illustration 3.7 Collaboration diagrams for some basic functionality of the transient event and detector data stores: (a) retrieving and storing event data objects, (b) synchronizing the transient detector store with the current event time.

3.11 Physical Design

For large software systems, such as ours, it is clearly important to decompose the system into hierarchies of smaller more manageable components.This decomposition of the system can have important consequences for implementation related issues, such as compile-time coupling, link-time dependencies, size of executables etc. The architecture must therefore take into account physical design issues as well as the logical structure.

Physical design focuses on the physical structure of the system. A physical component is the smallest unit of physical design and may contain several classes that together cooperate to provide some higher level functionality. However larger systems require hierarchical physical organization beyond the hierarchy of components. This can be achieved by grouping related components together into a cohesive physical unit, which we call a **package**. The notation used to represent packages follows that of Lakos [5] and is shown in Illustration 3.8. Dependencies between packages reflect the overall dependencies among the components comprising each subsystem.

A package is a collection of components that have a cohesive semantic purpose. It might also consist of a loosely coupled collection of low-level re-usable components, such as STL. In general the dependencies between packages are acyclic (cyclic dependencies should be avoided at all costs). Physically a package consists of a number of header files and a single library file.

Organizing software in terms of packages has several advantages:

- each package can be owned and authored by a single developer
- acceptable dependencies can be specified, and approved by the system architect, as part of the overall system design
- highly coupled parts of the system can be assigned to a single package which makes change management easier
- packaging aids incremental comprehension, testing and reuse

A design goal of physical design is to minimize the number of package dependencies (Illustration 3.8 gives an example) which is done for the following reasons:

- improves usability i.e. do not link huge libraries just to use simple functions
- reduces number of libraries that must be linked
- minimizes size of executable image

Dependencies can be minimized by repackaging components e.g. by escalating a component from a lower to a higher level.

Usability is enhanced by minimizing the number of header files that are exported. Header files are exported only if a client of the package needs access to the functionality provided by that component.

Physical design also addresses issues related to the management of the code repository and release mechanism. The directory structure directly supports the organization of packages and, the allocation of files to subdirectories depends on whether interfaces of components are public or private.

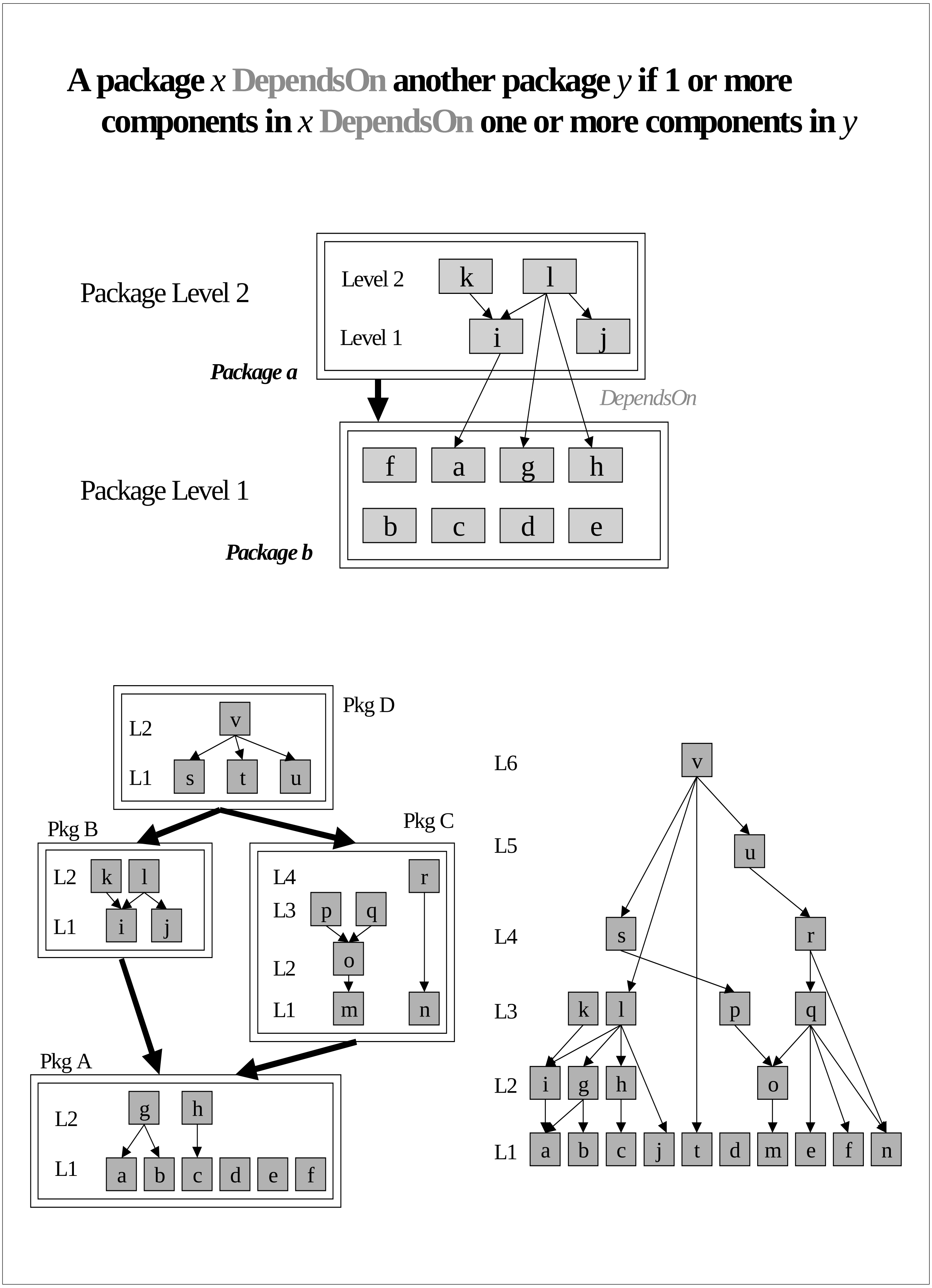


Illustration 3.8 Components, packages and their dependencies

4

Application Manager

The Application Manager (*ApplicationMgr*) is the component that steers the execution of the data processing application. There is only one instance of ApplicationMgr per application.

In this chapter we will describe the functionality of this component, the interfaces it offers to other components and its dependencies.

4.1 Purpose and Functionality

The main purpose of the Application Manager (*ApplicationMgr*) is to steer any data processing application. This includes all data processing applications for LHCb data at all stages: simulation, reconstruction, analysis, high level triggers, etc. Specific implementations of the ApplicationMgr will be developed to cope with the different environments (on-line, off-line, interactive, batch, etc.). The ApplicationMgr implements the following functionality:

- **The event loop**. For traditional batch processing applications it implements the "event loop". It initializes all the components and services of the application. In particular it initializes the event selector component with the selection criteria for the current job (which events need to be processed, etc.). Then it requests the processing services of all the relevant processing elements (*Algorithms*) for each event. Finally, it informs all the components when the job is done to allow them to perform the necessary actions at the end of job (saving histograms, job statistics, etc.).
- **Bootstraping the application**. The *ApplicationMgr* is in charge of creating all the services and processing components (*Algorithms*) needed to provide the desired functionality. The concrete type of these components depending on the persistent object store, user interface system, etc. is selected at run time based on the job options.
- **Service & Algorithm information center**. It maintains a directory of the major *Services* and *Algorithms* which have been created by it directly or indirectly. It allows other components to locate the requested service based on a name.

4.2 Interfaces

The *ApplicationMgr* provides the following interfaces:

- **Service locator (***ISvcLocator***)**. Algorithms and services may ask the *ApplicationMgr* for references to existing services belonging to the application. For example, to get a reference to the *MessageSvc* which is needed by the *Algorithm* in order to output an error message.
- **User interface (***IAppMgrUI***)**. Interface to the user interface to allow the user to interact with the application. This interface is necessary for non-batch oriented applications.
- **Properties interface** (*IProperty*). This interface allows other components to modify the default behavior of the *ApplicationMgr* by setting new values to properties. This interface is the same interface implemented by the *Algorithms* and other components.

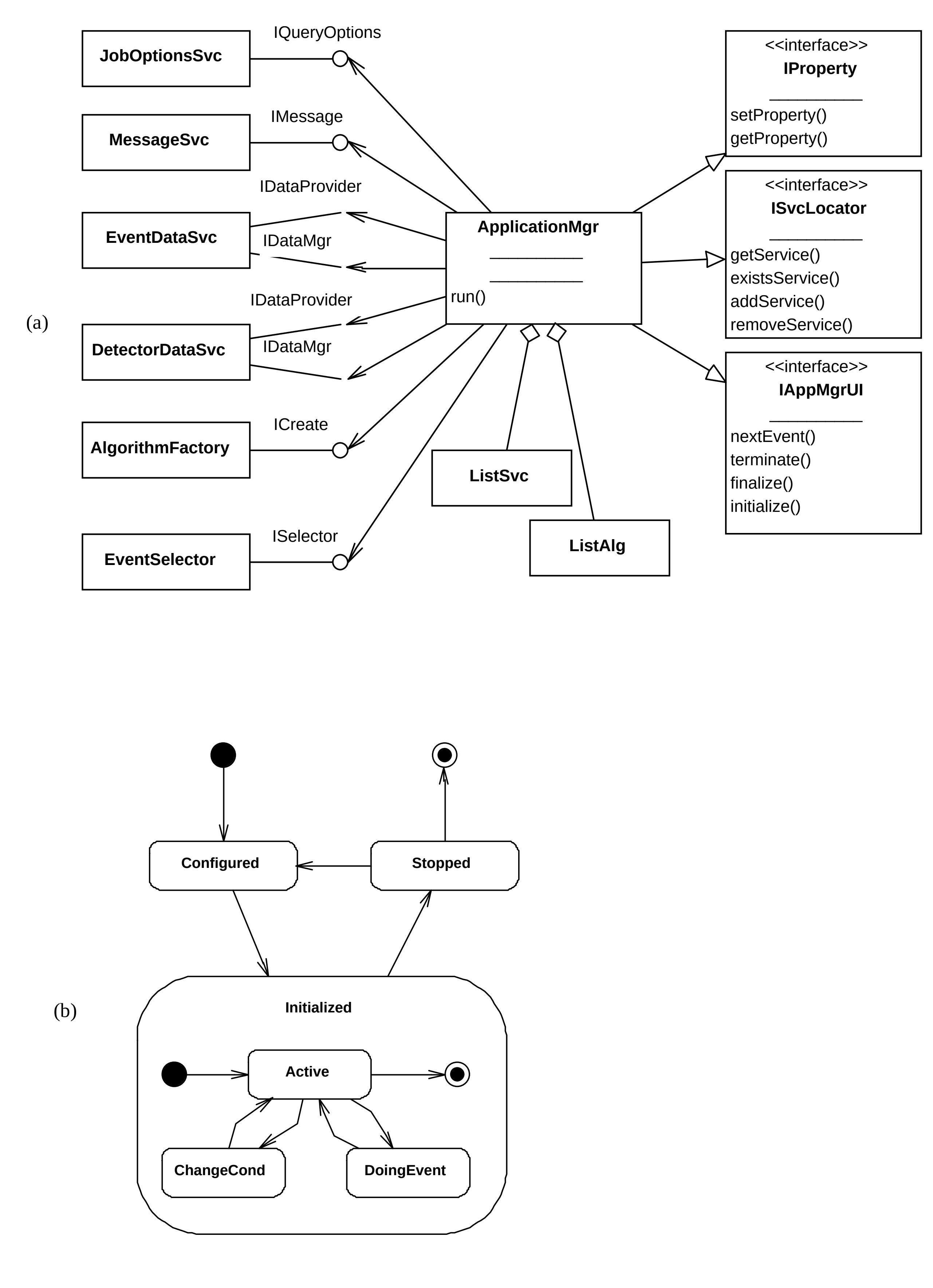


Illustration 4.1 The ApplicationMgr main class diagram (a) and state diagram (b)

4.3 Dependencies

The *ApplicationMgr* depends on the following services provided by other components of the architecture:

- Job options service (*JobOptionsSvc*). The *JobOptionsSvc* service provides to the *ApplicationMgr* the new values for its properties in case the user would like to overwrite the default ones. These properties may drive the choice of the persistent mechanism, the algorithms which need to be instantiated, the output stream for error messages, etc.
- Message service (*MessageSvc*). The *ApplicationMgr* uses the *MessageSvc* to report errors and informational messages to the end user.
- Algorithm factory (*AlgorithmFactory*). The *ApplicationMgr* uses the *AlgorithmFactory* service to create (instantiate) the concrete implementations of the *Algorithm*. In that way, the *ApplicationMgr* does not need to be changed when new algorithms are introduced into the system.
- Algorithms. Algorithms do the real physics data processing work.
- Event selector (*EventSelector*). The *ApplicationMgr* uses the event selector to generate the set of events which will processed by the application.

5

Algorithms

The *Algorithm* component is the support structure for real computational code. Any code that a user would like to be executed within the framework must conform to the Algorithm component specification[1]. This code may be for detector simulation, track reconstruction, calorimeter cluster finding, analysis of B-decays or whatever. In practice ensuring this conformity will be implemented by providing an Algorithm base class which must be extended to form concrete algorithms. A general algorithm has properties or parameters which tune the computation, one or more sources of input data, and one or more sources of output data.

In this chapter we describe the *Algorithm* base class from which all concrete algorithms must inherit. This interface allows the end-user to assemble and configure complex applications using basic algorithms as building blocks.

1 Except ofcourse for specialised things such as converters.

5.1 Purpose and Functionality

The purpose of a concrete algorithm is to convert a set of input data into a set of output data. The actual computation depends upon the specific algorithm and also upon the values of any internal parameters which may be set, for example, by the job options service.

Algorithms may only request data from the transient data services, e.g. the eventDataSvc, they know nothing of the persistent world. Similarly any data produced by an algorithm which is to be passed onto another algorithm or which is to be made persistent must be registed in one of these stores.

An algorithm may be executed once per event, or many times per event. For some types of algorithm it may be interesting to execute them only in response to certain events, e.g. when the run changes or when the current job is finished. Algorithms may be executed directly by the application manager or, if nested, by the parent algorithm.

5.2 Interfaces

The *Algorithm* provides the following interfaces:

- **Execution interface** (*IAlgorithm*). It is through this interface that algorithm configuration and execution is performed.
- **Properties interface** (*IProperty*). The Properties interface provides a hook to allow the setting of the internal attributes of a concrete algorithm

5.3 Dependencies

The *Algorithm* depends on the following services provided by other components of the architecture:

- Job options service (*JobOptionsSvc*). The *JobOptionsSvc* service provides to the *Algorithm* the new values for its properties in case the user would like to overwrite the default ones. These properties may change the internal behavior of the *Algorithm*.
- Event data service (*EventDataSvc*). The *Algorithm* used the EventDataSvc to get access to the data objects it needs to perform its function.
- Message service (*MessageSvc*). The *Algorithm* may need to send messages reporting its progress or errors occurring either for debugging or logging.
- Algorithm factory (*AlgorithmFactory*). The *Algorithm* uses the *AlgorithmFactory* service to create (instantiated) the concrete implementations of other *Algorithm*s.
- Algorithms. The *Algorithm* may use other smaller algorithms to implement its function.
- Other services like *EvtPersistecySvc*, *EventSelector*, etc. are initialized by the *ApplicationMgr* in

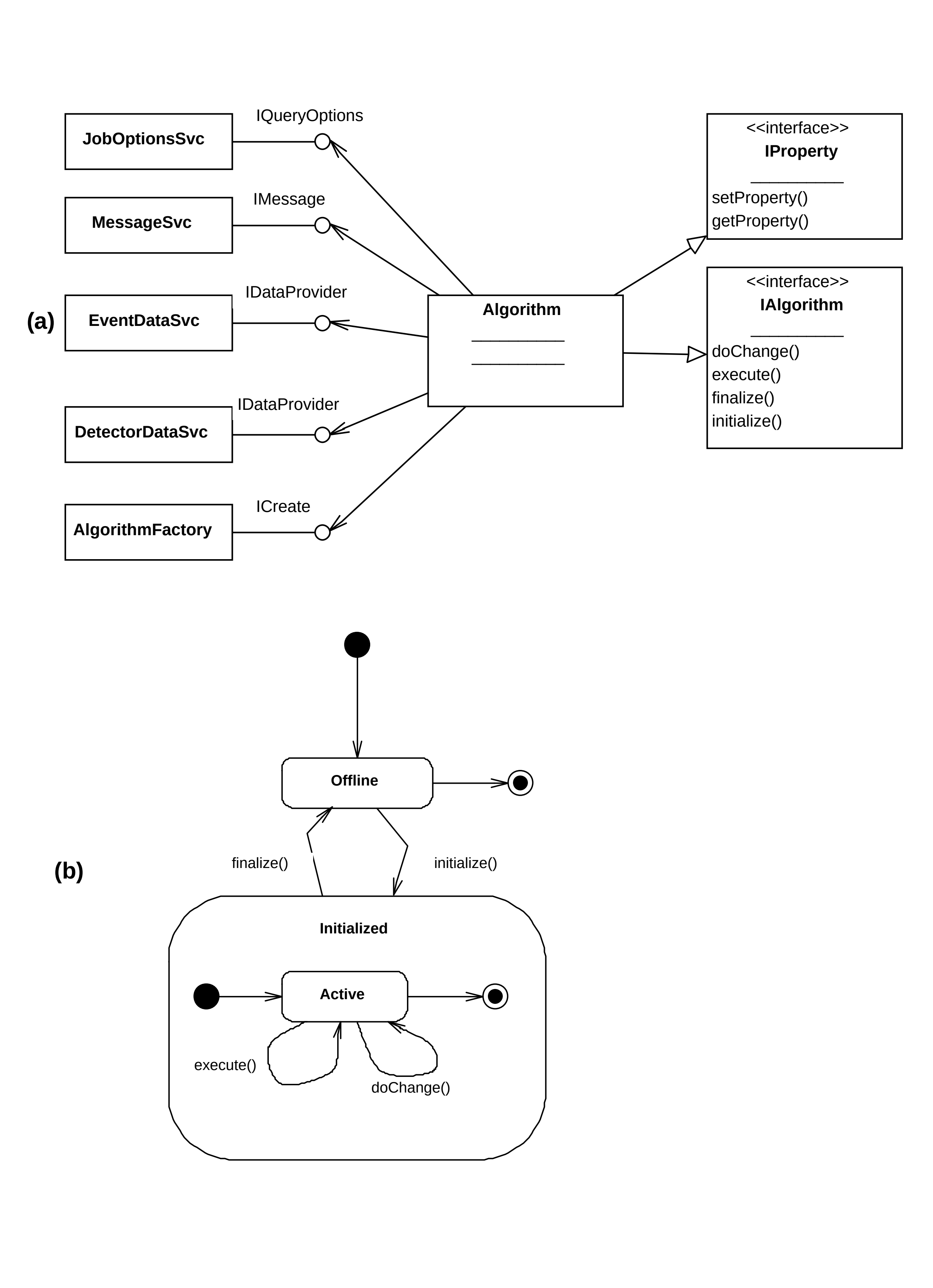


Illustration 5.1 (a) The Algorithm class diagram
(b) The Algorithm internal state diagram

6

Data Converter

A data *Converter* is responsible for translating data objects from one representation into another. Concrete examples are e.g. converters creating transient objects representing parts of an event from the persistent (and disk based) representations, or converters creating a textual representation of data objects for printing to the alphanumeric terminal. Specific *Converters* will be to be needed for each data type that needs to be converted. Any code that a user needs to execute for converting data objects from one representation into another must conform to the *Converter* component specification. In practice ensuring this conformity will be implemented by providing an *Converter* base class which must be extended to form concrete converters.

In this chapter we describe the *Converter* base class from which all concrete converters must inherit.

6.1 Purpose and Functionality

The data converters are responsible for translating data from one representation into another. Concrete examples are e.g. converters creating transient objects from their persistent (disk based) representations. Converters will have to deal with the technology both representations are based on: in the upper example they have to know about the database internals as well as the structure of the transient representations. The converters know about the mechanism to retrieve persistent objects (ZEBRA, Objectivity, …) and only pass abstract instances of the converted objects, hence shielding the calling service from internals.

Data converters are meant to be light. This means there will not be all-in-one converters, which are able to convert the “world”, but rather many converters, each able to create a representation of a given type.

In order to function a converter must be able to

- Answer (when asked) which kind of representation the converter is able to create and on which kind of data store the source representation of the object resides.
- Retrieve the source object from the source store.
- Create the requested transient representation using the information contained in the source object.
- Initialize pointers in the transient representation of the created object.
- Update the transient representation using the information contained in the source object.
- Update pointers in the transient representation of the created object.
- Convert the transient representation to its target (e.g. persistent) representation.
- Resolve references within the target representation (persistent references).
- Update the target representation from the transient representation.
- Update references within the target representation (persistent references).

The conversion/creation mechanism of an object into another representation is a two step process:

- Firstly the raw object will be translated. This does not include any links pointing to other objects.
- At the second step the link will be converted.

Concrete user converters are based on a base class which deals with the technology specific actions. The concrete converters hence only deal with data internal to the objects, e.g.

- Resolving pointers of the transient objects to objects in the detector description.
- Fill/update contained entries in the persistent representation which will not be identifiable by the persistent store.
- The base class is in charge of resolving the generic “identifiable” references as they are accessible from the directory of the transient *DataObject*. It is understood, that there is a correspondence between the identifiable entries e.g. in the transient world and the persistent world.

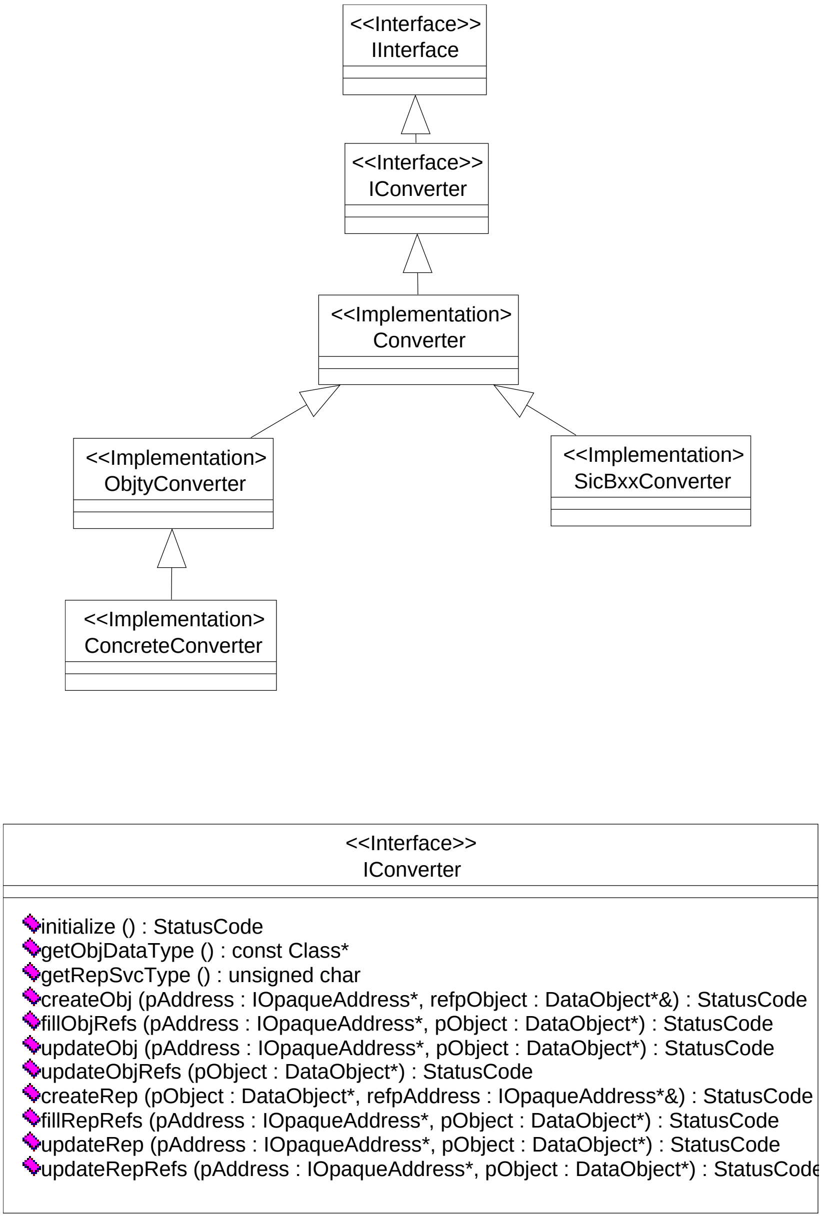


Illustration 6.1 The Converter class diagram with a possible layering of increased functionality: ObjtyConverter and SicBxxConverter can already handle the abstract data model leaving only “primitives” to the concrete converter. Below the provisional definition of the interface as it is known to the calling services.

6.2 Interfaces

IConverter: The interface allows the calling services to pass the necessary information to the converter and to interact without coupling to internals.

6.3 Dependencies

The *Converter* depends on the following services provided by other components of the architecture:

- The generic Converter implementation (*Converter)* offers some standard functionality and eases n the implementation of specific converters.
- Message service (*MessageSvc*). The *Converter* uses the *MessageSvc* to report errors and informational messages to the end user.
- The data storage technology: Converters will have to deal with specific data stores like ZEBRA, Objectivity etc.

7

Job Options Service

The purpose of the job options service is to supply the options for the current job to other components of the architecture. It is assumed that facilities allowing the user to edit the options and to save or retrieve sets of them for future use are supplied outside this service.

7.1 Purpose and Functionality

The purpose of the job options service is to supply the options for the current job to other components of the architecture. It is assumed that facilities allowing the user to edit the options and to save or retrieve sets of them for future use are supplied outside this service.

The options may consist of:

- Flags that select the algorithms to be run in the current job.
- Values that override default properties of the algorithms
- Definition of the input data set
- Selection criteria for the input data
- Definition of output data streams (e.g. for event data, histogram data etc.)
- Definition of error reporting streams
- Etc.

A set of options is identified by the name of the set (*SetName*) (For example *SetName* could be the full pathname of a file if the options are contained in a text file, or the identifier of a set in the options database)

An option modifies the properties of a client. Each option consists of:

- The name of the client (*ClientName*) to whose properties will be modified by this option.
- The name of the property to be modified (*PropertyName*)
- The new value (or values) of the property (*PropertyValues*) (T = bool, int, double, string)

7.2 Interfaces

The *JobOptionsSvc* provides the following interfaces:

- **Query options interface** (*IQueryOptions*). This interface exposes the functionality of the *JobOptionsSvc*. It provides the following methods:
    - *configure(string SetName)* This method allows a client to specify which set of job options is to be used in the current context. Usually this will be used only once per job when *JobOptionsSvc* is created.
    - *setMyProperties(IProperty *me, string ClientName)* This method is used by a client who wants to override the default values of its properties with those described in the job options. The client must implement the *IProperty* interface, which will be used by setMyProperties to set the client's properties.
    - *getOption(string ClientName, string PropertyName, vector<T> PropertyValues)* This method allows clients to retrieve the job option which modifies a given property of a given client in the current context, if that option exists.

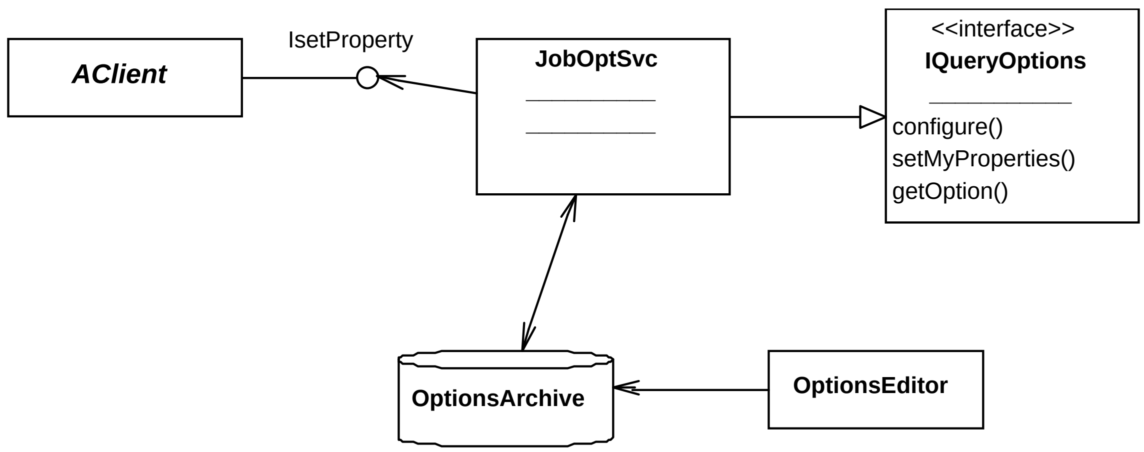


Illustration 7.1 The JobOptionsSvc class diagram

7.3 Dependencies

A given *PropertyName* can occur only once for a given *ClientName*. This implies that different instances of a client class must have a different *ClientName* if they need different *PropertyValues* for a given *PropertyName*.

The *JobOptionsSvc* depends on the following services provided by other components of the architecture:

- *ApplicationMgr*. The context must be initialized by an external client. It is assumed that the SetName is provided by the user e.g. as an input argument to main() or as an environment variable.
- *IProperty* interface. Clients wishing to use the setMyProperties method must implement this interface.
- OptionsArchive. An archive is needed for previously defined sets of options. For example this could be a set of text files, or a database.
- OptionsEditor. An editor is needed to edit the set of options. For example this could be a text editor, or a database editor.

8

Event Selector

The event selector allows an end user to select, on the basis of physical properties, which events will be processed by an application. The event selector is the component which knows what is the next event to be processed.

In this chapter we will describe the functionality envisaged for this component, the interfaces it offers to other components and its dependencies.

8.1 Purpose and Functionality

The event selector (*EventSelector*) component is able to produce a list of events from a given set of "selection criteria". In general, for batch oriented applications, it is the *ApplicationMgr* that provides the "selection criteria" to the *EventSelector.* For interactive applications, it is required that the end user has the possibility to interact directly with the *EventSelector* by means of the *UserInterface* component. The complexity of the "selection criteria" can vary from very simple to very sophisticated involving looking at the event data and running some selection code to decide whether the event is selected or not. Here are some examples of "selection criteria":

- All the events of a given run number or within a range of run numbers.
- All the events between two dates that belong to a certain event classification.
- A discrete list of run number, event number pairs.
- All the events of a given persistent "event collection" identified by a name.
- A complex SQL query in the event data ODBMS.
- etc.

The *EventSelector* provides one or more iterator types to be used by the *ApplicationMgr.* It is the role of the *ApplicationMgr* to enquire of the *EventSelector* which is the next event to be processed and to setup correctly the corresponding data stores with that information. The *EventSelector* does not necessarily need to have in memory all the handles of the selected events since event selections could be very big.

It is not the role of the *EventSelector* to create persistent "event collections". These collections are created by specialized applications called "event selection applications". The purpose of these applications is to processes events from some general collections (i.e. runs), apply physics *Algorithms* and select only those events that pass all the physics selection criteria, and finally to create an "event collection" as output.

Responsibilities

- The *EventSelector* must be capable of examining data from any of the possible sources e.g. data stored in an Objectivity database, data stored in ZEBRA format on tape etc.
- The *EventSelector* will accept user defined selection function objects.

8.2 Interfaces

The *EventSelector* provides the following interfaces:

- Event selection interface (*IEvtSelector*). This interface is used to tell the selector what is the selection criteria, declare user selection functions, and obtain references to iterators.
- Selection user interface (*ISelectorUI*). This is the interface used by the *UserInterface* component which allows the end user to interact with the selector.
- Properties interface (*IProperty*). This interface allows other components to modify the default behavior of the EventSelector by assigning new values to its properties.

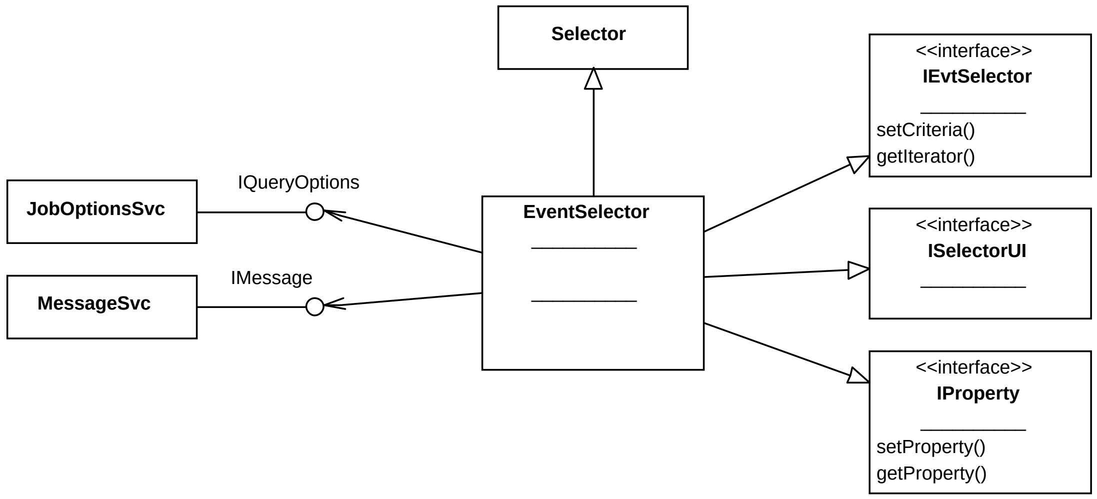


Illustration 8.1 The EventSelector class diagram

9

Transient Data Store

A transient data store is a passive component which acts as the “logical” storage place for transient data objects.

9.1 Purpose and Functionality

We envisage several transient data stores within one application, each characterized by the lifetime and nature of the contained data. For example, there is the Transient Event Data Store for event data, the Transient Detector Data Store for detector, alignment and calibration data, and the Transient Histogram Data Store for histograms and other “statistics based” objects.

A Transient Data Store is managed by a Data service, for example the Transient Event Data Store is managed by the Transient Event Data Service (EventDataSvc). This service implements an interface which is used by clients and in particular by Algorithms to access the data within the store. In order to find required data it is assumed that a client knows how the data is organized, and identified and obeys the relevant ownership conventions. Key issues related to transient stores are:

- Data Organization. The data within a data store is organized as tree, see Section 3.4.
- Data Identification. Any data object in the data store has an identifier, allowing it to be refered-to by a client [Section 3.4].
- Data ownership. Data objects within a data store are *owned* by the data service that is managing that store. In particular if a new data object is created by a client and registered into the store so as to make it available to other algorithms then that client is no longer responsible for the deletion of the object. Infact after registration they must not delete the object or change essential characteristics such as the identification, etc..
- Lifetime. References to objects in the data stores are valid only during a specific (store dependent) time span. For example, references to event data objects become invalid immeadiately the next event is loaded.

9.2 Interfaces

The *Transient Data Store* does not offer any interface directly to clients. The interface is implemented by the corresponding *Data Service*. Clients of a *Transient Data Store* are allowed to keep direct references to objects in the store as long as the validity period and ownership conventions are obeyed.

9.3 Dependencies

Since the Transient Data Store is a passive component it does not need any other component of the system to perform its function (storing data objects is memory).

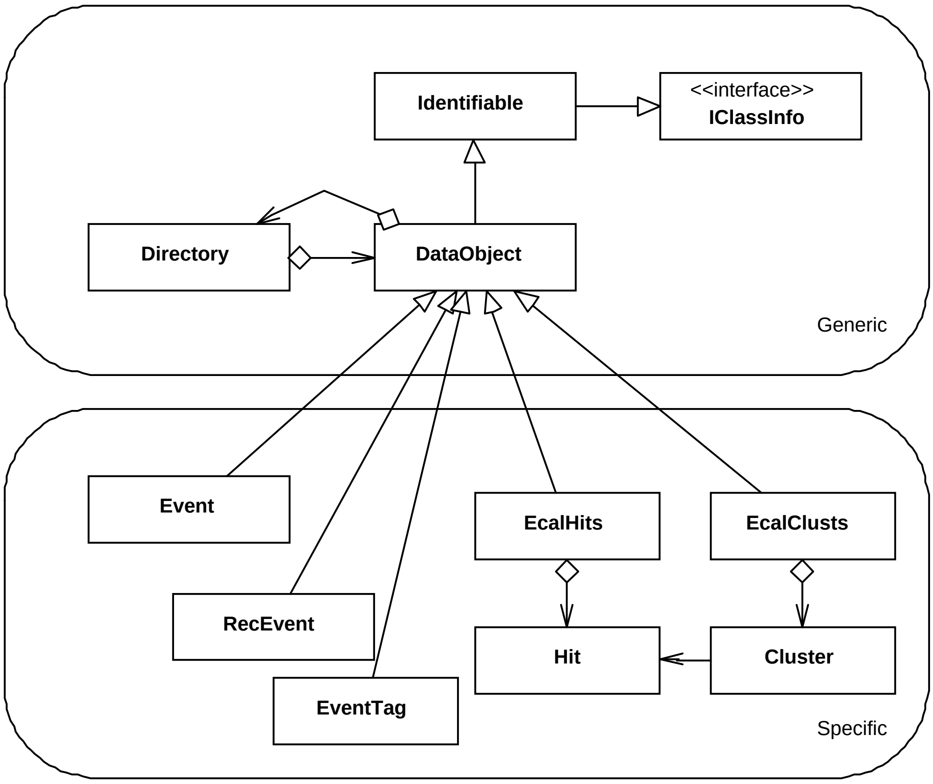


Illustration 9.1 The class diagram for the *Data Objects* in the store.

10

Event Data Service

The event data service manages the transient event data store. This component provides the necessary functionality required by algorithms and the application manager to locate data object within the transient store and to register new ones.

10.1 Purpose and Functionality

The event data service manages the transient event data store. The service interacts mainly with two components: *Algorithms* and the *ApplicationMgr.*

- The event data service is used by the *Algorithms* as an input/output channel for data objects. An algorithm may request objects from the store, or register them into the store so that they are available to other algorithms.
  - The event data service delivers references to event data objects on request. If the requested data objects are not present, the event data service asks the event persistency service to load the required objects and then makes them availible to the algorithm.
  - Once event data objects are registered to the event data service, the algorithm gives up ownership. The event data service releases the objects on request of the application manager.
- Event data objects must be identifiable in order to be added to the data store.
- The registration of data must respect the tree structure of the transient store.
- The application manager tells the event data service which event to deal with in case of a request.
- Clients of the event data service can decide which data should be made persistent:
  - Clients may decide to discard partially or completely the data objects managed by the service.
  - The event data service must be able to deliver transient data objects to the services in charge of creating other data representations like the persistency service or the service responsible for creating graphical representations.

10.2 Interfaces

The *EventDataSvc* implements the following interfaces:

- **Generic service interface** (*IService*) for specific interaction like e.g. query the service name.
- **Storage management** (*IDataManagerSvc*): This interface supplies global management actions on the transient data store. This interface is used by the *ApplicationMgr.* Through this interface actions necessary to manipulate event related data globally, such as requests to load a new event, discard the objects owned by the service etc. are handled. The interface also allows selection collectors using selection agents to traverse through the data store.
- **Data provider** (*IDataProviderSvc*): The interface used by the *Algorithms* to request or register event data objects.

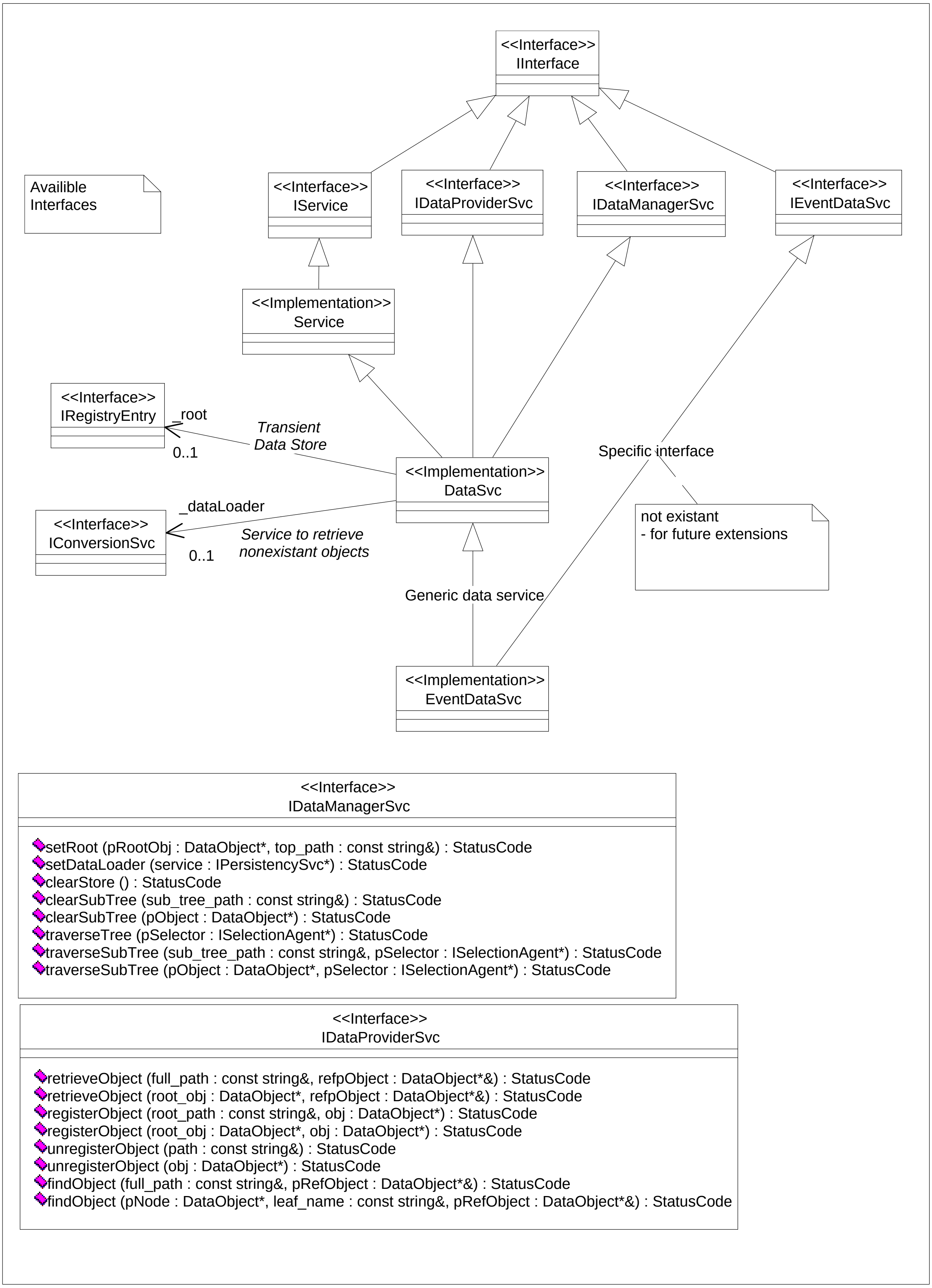


Illustration 10.1 The EventDataSvc class diagram and the provisional defintion of the interfaces.

10.3 Dependencies

The *EventDataSvc* depends on the following services provided by other components of the architecture :

- The generic service (*Service*) implementation which defines basic properties and abilities of any service.
- Job options service (*JobOptionsSvc*). The *JobOptionsSvc* service provides to the *EventDataSvc* the new values for the its properties in case the user would like to overwrite the default ones. The implementation of the event data service only depends on the interface of the job option service.
- Message service (*MessageSvc*). The *EventDataSvc* uses the *MessageSvc* to report errors and informational messages to the end user. The implementation of the event data service only depends on the interface of the message service.
- Event persistency service (*EvtPersistencySvc*). The *EventDataSvc* uses the persistency service to create transient objects on demand. The implementation of the event data service only depends on the interface of the persistency service.
- Data selection agents (*ISelectionAgent*). Data selector agents are used to traverse the data store in order to analyse the content e.g. to collect references to objects which should be passed to other services.

11

Event Persistency Service

The event persistency service delivers data objects from a persistent store to the transient data store and vice versa. The persistency service collaborates with the event data service to provide the data requested by an algorithm in the case that the data is not yet in the transient store. This service requires the help of specific converters which actually perform the conversion of data objects between their transient and persistent representations.

11.1 Purpose and Functionality

The purpose of the event persistency service (*EvtPersistencySvc*) is the extraction of persistent objects from the persistent data store and their conversion to the appropriate transient types, and the inverse process.

The creation of a transient object proceeds as follows:

- given an identifier, the service locates the object in the persistent world,
- the appropriate *Converter* for this type is selected,
- the persistency service invokes the converter and finally
- delivers the requested transient object to the client.

The object identifier needs only to be interpreted by the *Converter*, it is not needed anywhere outside the converter.

If it is known in advance that not a single object, but instead a complete tree or part of a tree of data is required, then the service can optimize the procedure. For example, sub-trees or a list of objects required in every event may be specified to the persistency service and the objects “pre-loaded” before they are requested by an algorithm.

To populate the persistent data store is slightly different:

- the persistency service is given transient objects,
- it finds the proper converter for the received object,
- it invokes the object conversion using this converter and
- stores the persistent object.,

Converters being able to create persistent representations must be assigned to the service either one by one or in the form of a factory.

Converters are declared to the *EvtPersistencySvc* at run-time and therefore they need to implement a common interface. The *Converters* know about the mechanism to retrieve persistent objects (e.g. from ZEBRA, Objy,...) and to create their transient representation and vice-versa. Identifying which *Converter* is able to convert a specific instance is achieved by associating each *Converter* to a unique run-time independent class identifier.

11.2 Interfaces

The *EvtPersistencySvc* provides the following interfaces:

- Generic service interface (*IService*) for interactions such as querying the service name.
- Generic data conversion interface (*IConversionSvc*). This interface is capable to accept converters or a converter factory necessary to create persistent event data representations. This interface also accepts data selectors which tell the service which objects have to be created, converted or updated.
- Persistency specific interface (*IPersistencySvc*). This interface will handle - if needed - specific interactions with the persistent store.

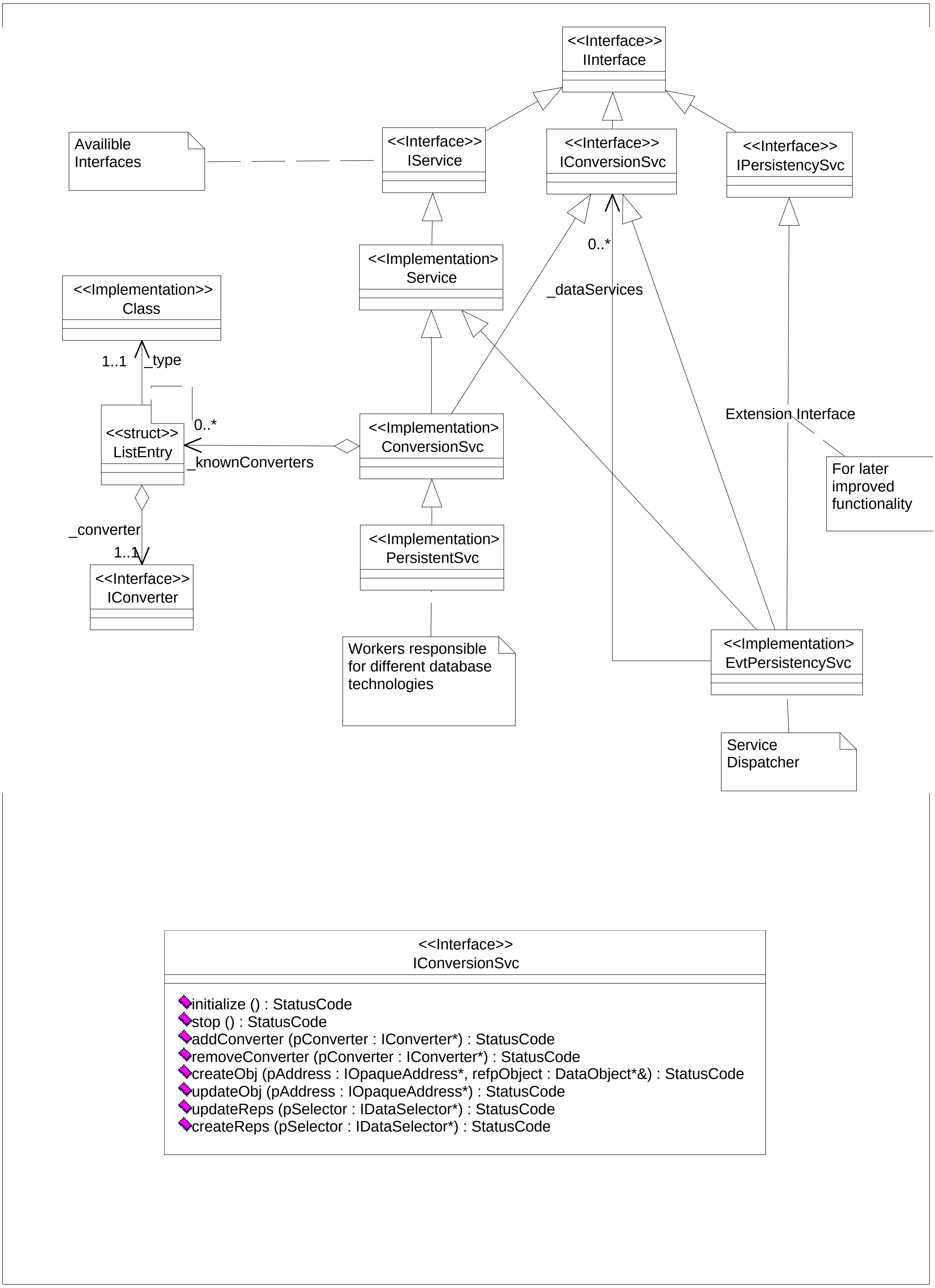


Illustration 11.1 The EvtPersistencySvc and the provisional interface.

11.3 Dependencies

The *EvtPersistencySvc* depends on the following services provided by other components of the architecture:

- The generic service (*Service*) implementation which defines basic properties and abilities of any service.
- Job options service (*JobOptionsSvc*). The *JobOptionsSvc* service provides to the *EvtPersistencySvc* the new values for the its properties in case the user would like to overwrite the default ones. The implementation of the event data persistency service only depends on the interface of the job option service.
- Message service (*MessageSvc*). The *EvtPersistencySvc* uses the *MessageSvc* to report errors and informational messages to the end user. The implementation of the event data persistency service only depends on the interface of the message service.
- Event data service (*EventDataSvc*). The *EvtPersistencySvc* collaborates very closely with the *DetPersistencySvc* to load or update detector objects in to store. The implementation of the event data persistency service only depends on the interfaces of the data service.
- Data item selectors (*DataItemSelector)*. If persistent data must be updated or new persistent objects must be created an object selection is passed to the service in form of a selected item collection. The service will retrieve all selected items from the selector.

12

Detector Data Service

The detector data service manages the transient detector data store. This component is very similar to the event data service with regards to the retrieval and registration of data objects. However in addition it is responsible for the management of such things as the synchronization of detector information to the current event.

12.1 Purpose and Functionality

The detector data service manages the transient detector data store. The service interacts with several other components: the application manager, algorithms and event data converters whenever event data structures have references to detector data.

Besides providing requested data, the detector data service also has to check the validity of the data since detector data are not “static” over the entire lifetime of a job. For example calibration constants will change with time and the set of constants which is valid for the first events in the job may not be valid for last events. Hence the detector data service must be able to distinguish between valid and invalid data with respect to a given event.

Invalid data may be discarded or updated, i.e. given a valid time stamp the detector data service must:

- direct new detector data requests to the proper, valid data objects.
- request the loading of up-to-date detector data objects if they are not present in the store.

Algorithms will initialize references to detector data at configuration time. These references must stay valid over the analyzed event range and so the objects will have to be updated rather than recreated.

This means, that the detector data service is not stateless. It will have to be validated for every event ensuring that the contained references are valid. In order to optimize data validity the detector data objects which should automatically be updated must be marked.

- The detector data service is used by the algorithms as an input/output channel for detector data objects:
    - The detector data service delivers references to detector data objects on request. If the requested data objects are not present, the detector data service asks the detector persistency service to deliver the objects and makes them available to the algorithm.
    - Detector data objects intended to be available to (other) algorithms must be registered to the detector data service.
    - Once detector data objects are registered to the detector data service, the algorithm gives up ownership. The detector data service releases the objects at the request of the application manager.
- Detector data objects must be identifiable in order to be added to the data store.
- The registration of data must follow the hierarchy of the detector data objects.
- Clients of the detector data service can decide which data should be made persistent:
- Clients may decide to discard partially or completely the data objects managed by the service.
- The detector data service must be able to deliver transient data objects to the services in charge of creating other data representations like the persistency service or the service responsible for creating graphical representations

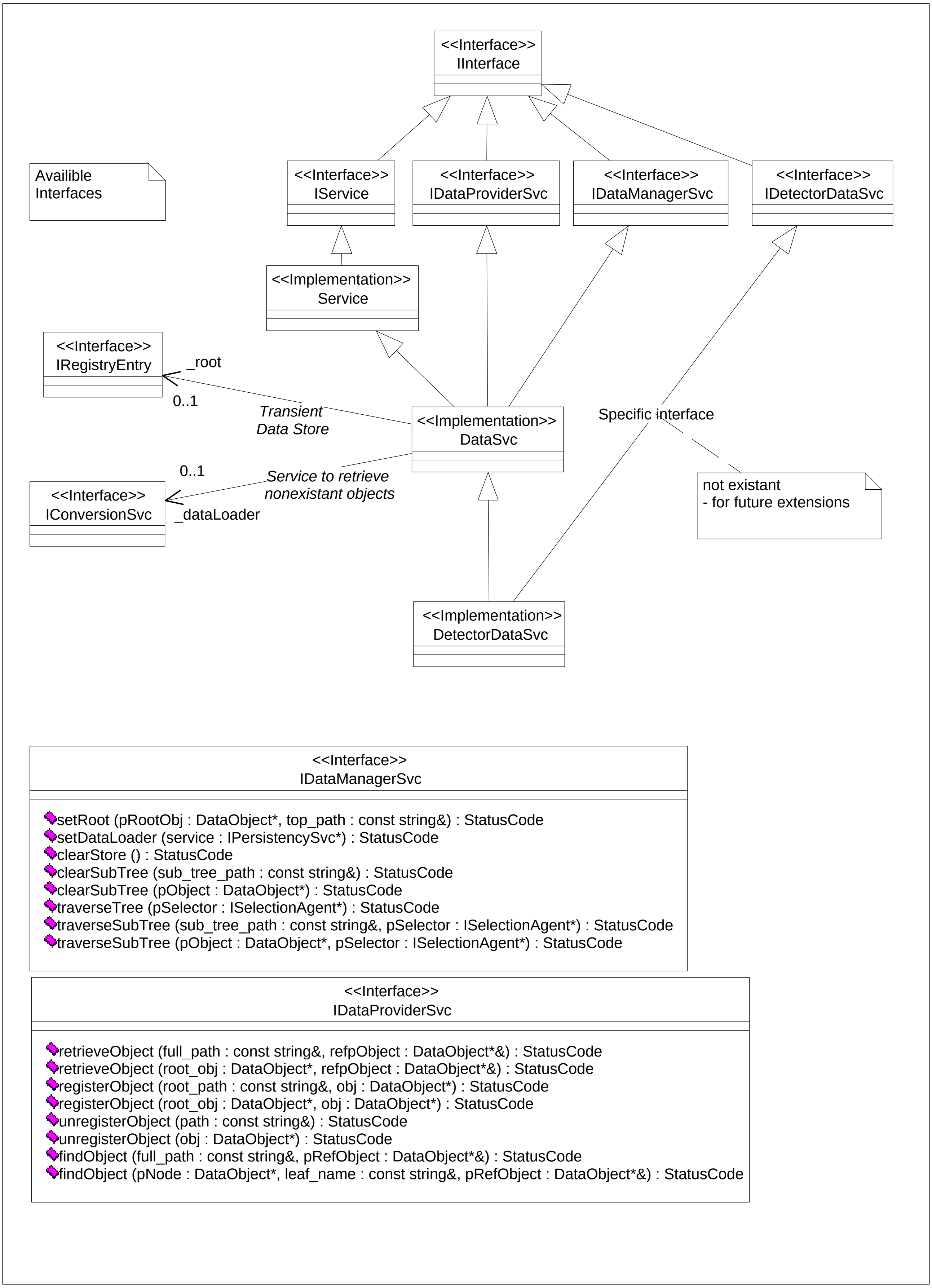


Illustration 12.1 The DetectorDataSvc class diagram and the provisional definition of the interfaces.

12.2 Interfaces

The *DetectorDataSvc* implements the following interfaces:

- **Generic service interface** (*IService*) for such things as querying the service name.
- **Storage management** (*IDataManagerSvc*). This interface is used primarily by the *ApplicationMgr* to manipulate globally detector related data, e.g. to initialize a new detector store, to discard the objects owned by the service etc. The interface also allows selection collectors using selection agents to traverse through the data store.
- **Data provider** (*IDataProviderSvc*). The interface is used by the *Algorithms* to request or register detector data objects.

12.3 Dependencies

The *DetectorDataSvc* depends on the following services:

- The generic service (*Service*) implementation which defines the basic properties and abilities of any service.
- Job options service (*JobOptionsSvc*). The *JobOptionsSvc* allows a user to override the default properties of the service.
- Message service (*MessageSvc*). The *DetectorDataSvc* uses the *MessageSvc* to report errors and informational messages to the end user.
- Detector persistency service (*DetPersistencySvc*). The *DetectorDataSvc* collaborates very closely with the *DetPersistencySvc* to load or update detector data objects.
- Data selection agents (*ISelectionAgent*). Data selector agents are used to traverse the data store in order to analyse the content e.g. to collect references to objects which should be passed to other services.

13

Detector Persistency Service

The detector persistency service delivers transient detector data objects which originally reside in a persistent store. This service will need to know the model of how different versions of the detector description, calibration and alignment are stored.

13.1 Purpose and Functionality

The purpose of the detector persistency service is the conversion of persistent detector data objects into transient detector data objects and vice versa. Detector data covers all non event-related data which is necessary to interpret the event data properly. This consists of geometry data and detector conditions (calibration data etc.). Usually the lifetime of this data spans many events.

As regards the conversion between transient and persistent types the *DetPersistencySvc* behaves just as the *EventPersistencySvc* does, but additionally:

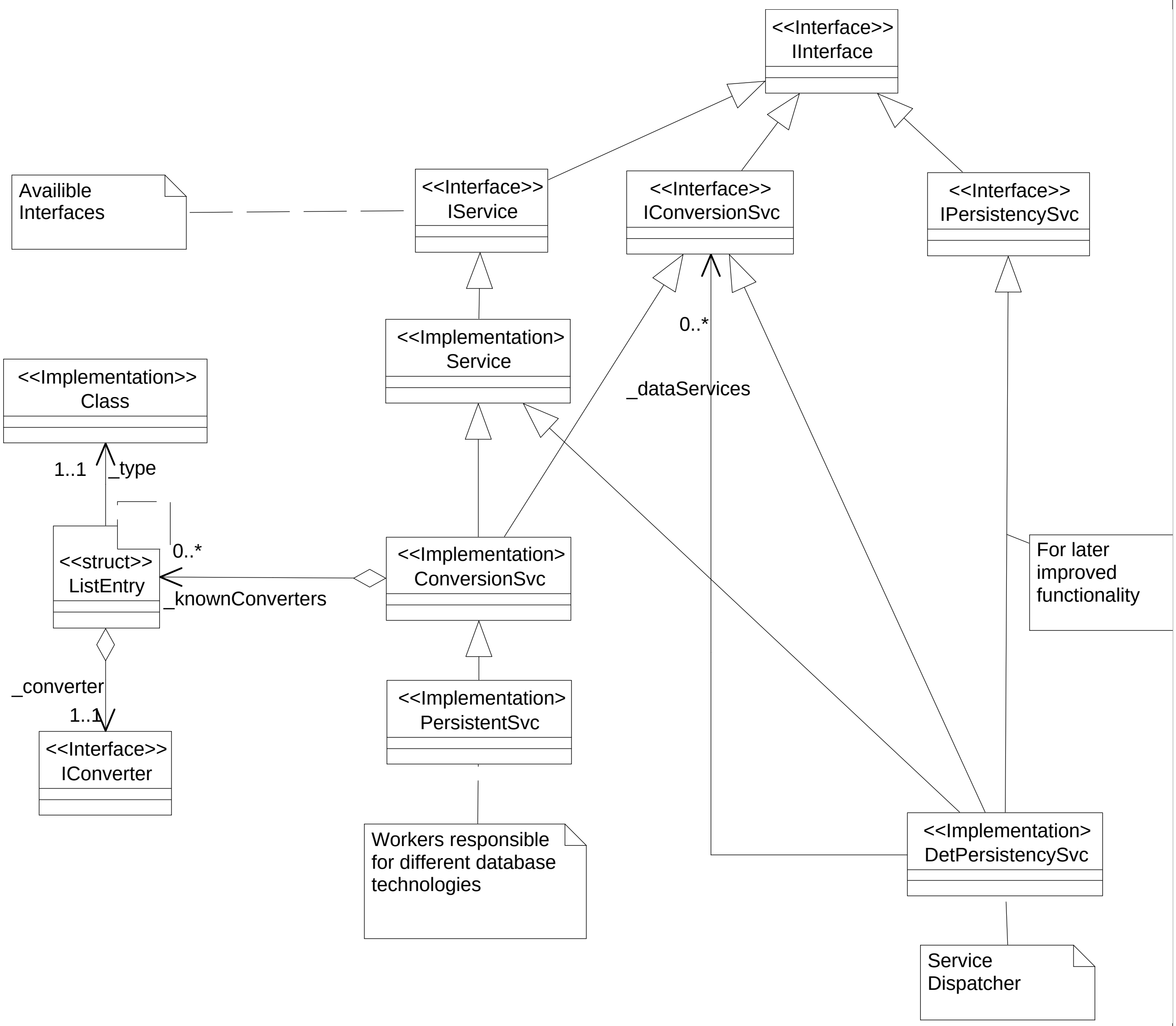
- The service will delegate object updates to the corresponding data converter.

Converters are given to the Persistency service at run-time and hence have to provide a common interface in order to be useful to the persistency service. The converters know about the mechanism to retrieve persistent objects (e.g. from ZEBRA, Objty,...) and pass created transient objects to the service. For a more detailed description of generic data persistency service functionality please see the description of the *EventPersistencySvc*.

13.2 Interfaces

The *DetPersistencySvc* provides the following interfaces:

- Generic service interface (*IService*)
- Generic data conversion interface (*IConversionSvc*). This interface is able to accept converters or a converter factory necessary to create persistent detector data representations. This interface also accepts data selectors which tell the service which objects have to be created/converted/updated.
- Persistency service interface (*IPersistencySvc*). This interface will handle - if needed - specific interactions with the persistent store.



<<Interface>> IConversionSvc
initialize () : StatusCode stop () : StatusCode addConverter (pConverter : IConverter*) : StatusCode removeConverter (pConverter : IConverter*) : StatusCode createObj (pAddress : IOpaqueAddress*, refpObject : DataObject*&) : StatusCode updateObj (pAddress : IOpaqueAddress*) : StatusCode updateReps (pSelector : IDataSelector*) : StatusCode createReps (pSelector : IDataSelector*) : StatusCode

Illustration 13.1 The DetPersistencySvc class diagram and the provisional interface.

13.3 Dependencies

The *DetPersistencySvc* depends on the following services provided by other components of the architecture:

- The generic service (*Service*) implementation which defines basic properties and abilities of any service.
- Job options service (*JobOptionsSvc*). The *JobOptionsSvc* service provides to the *DetPersistencySvc* the new values for the its properties in case the user would like to overwrite the default ones. The implementation of the detector persistency data service only depends on the interface of the job option service.
- Message service (*MessageSvc*). The *DetPersistencySvc* uses the *MessageSvc* to report errors and informational messages to the end user. The implementation of the detector persistency data service only depends on the interface of the message service.
- Detector data service (*DetectorDataSvc*). The *DetectorDataSvc* collaborates very closely with the *DetPersistencySvc* to load or update detector objects in to store. The implementation of the detector persistency data persistency service only depends on the interfaces of the data service.
- Data item selectors (*DataItemSelector)*. If persistent data must be updated or new persistent objects must be created an object selection is passed to the service in form of a selected item collection. The service will retrieve all selected items from the selector.

14

Histogram Data Service

The main purpose of the histogram service is to add histograms or other statistical data to the transient histogram store and to retrieve them when necessary. This component is very similar to the event data service with regards to the retrieval and registration of data objects.

14.1 Purpose and Functionality

The main purpose of the Histogram Service is to manage the *transient histogram store* and in particular to add or create histograms or other statistical objects and to retrieve them when necessary. It is servicing mainly the *Algorithms*.

The service is responsible for the following:

- Support for the creation (booking) of histograms in the transient data store.
- Registering histograms created outside the transient data store.
- Searching for histograms in the transient data store.
- Deletion of histograms in the transient data store.
- Forwarding requests to the *HistoPersistencySvc* for saving and retrieving histograms from persistent storage.

The *transient histogram store* should be a (specialized) instance of a more generic transient data store. It is a separate store mainly because the lifetime of the objects it contains is usually the complete job. By default the *ApplicationMgr* does not perform managerial actions at regular intervals as is the case for the *transient event store*.

14.2 Interfaces

The *HistoDataSvc* provides the following interfaces:

- **Storage management** (*IDataManager*). This interface supports the global management of the transient data storage such as initializing it. This interface is used by the *ApplicationMgr.*
- **Data provider** (*IDataProviderSvc*). The interface used by the *Algorithms* to request or register histograms or other statistical objects.
- **Histogram service** (*IHistogramSvc*). This interface is used to provide histogram specific support to the *Algorithms* in terms of creating/booking and manipulating them. In this way we can isolate the underlying histogram packages from the *Algorithms*.

14.3 Dependencies

The *HistoDataSvc* depends on the following services provided by other components of the architecture:

- Job options service (*JobOptionsSvc*). For setting the properties of the *HistoDataSvc*. These properties may drive the choice of the persistent mechanism, the output stream for error messages, etc.
- Message service (*MessageSvc*). The *HistoDataSvc* uses the *MessageSvc* to report errors and informational messages to the end user.

Some of the functionality of the service has to do with the isolation of the *Algorithms* from a particular histogram technology. In any case the implementation of the service will depend on specific packages. Currently the options are: HBOOK (CLHEP) or HISTOOGRAM (LHC++).

15

Histogram Persistency Service

The histogram persistency service is responsible for storing histograms and N-tuples.

15.1 Purpose and Functionality

The histogram persistency service is responsible for storing histograms and N-tuples. The demands to the persistent histogram store are not much different than for other stores - except that this service will from individual physicists be rather used to write data rather than read data.

Writing N-tuples essential for physicists to refine data sets is very similar to writing event data. N-tuples are different from histograms: N-tuples read and analysed later must support navigation to source data and object linking. N-tuple entries without this functionality are of limited use.

Histograms usually can easily be transformed to flat floating arrays and hence be treated as any other variable size data type. For this reason no extra demands arise from the persistent representation of a histogram. The main difference results from the storage area: histograms will be produced and stored by many users - the results should be stored locally to optimize later access to the results.

15.2 Interfaces

The *HistoPersistencySvc* provides the following interfaces:

- Generic service interface (*IService*) for specific interaction like e.g. query the service name.
- Generic data conversion interface (*IConversionSvc*). This interface is capable to accept converters or a converter factory necessary to create persistent histogram data representations. This interface also accepts data selectors capable to tell the service which objects have to be created/converted/updated.
- Persistency service interface (*IPersistencySvc*). This interface will handle - if needed - specific interactions with the persistent store.

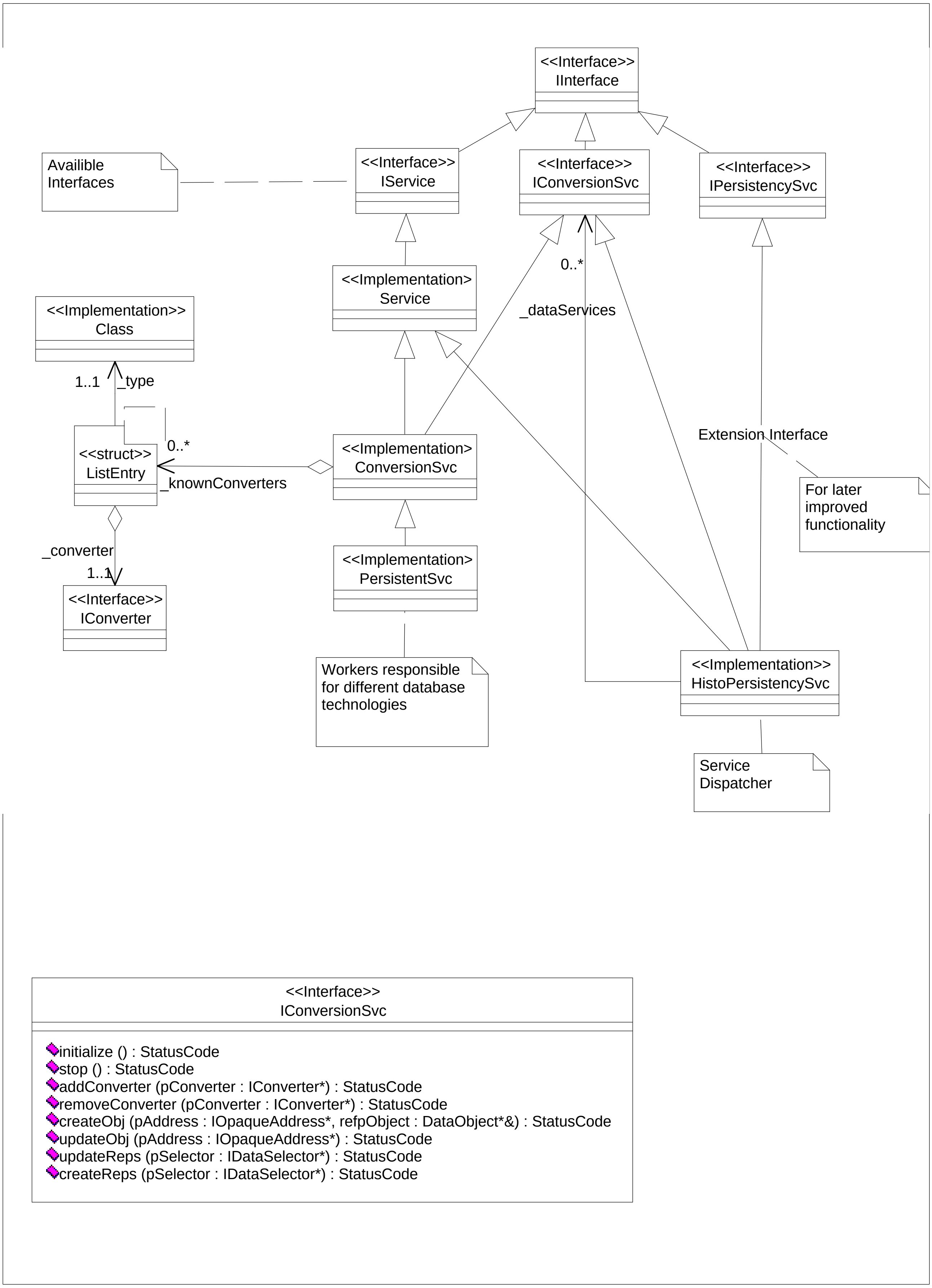


Illustration 15.1 The HistoPersistencySvc class diagram and the provisional interface.

15.3 Dependencies

The depends on the following services provided by other components of the architecture :

- The *HistoPersistencySvc* generic service (*Service*) implementation which defines basic properties and abilities of any service.
- Job options service (*JobOptionsSvc*). The *JobOptionsSvc* service provides to the *HistoPersistencySvc* the new values for the its properties in case the user would like to overwrite the default ones. The implementation of the depends only on the interface of the job option service.
- Message service (*MessageSvc*). The *HistoPersistencySvc* uses the *MessageSvc* to report errors and informational messages to the end user. The implementation of the service only depends on the interface of the message service.
- Histogram data service (*HistoDataSvc*). The *HistoDataSvc* collabotates very closely with the *HistoPersistencySvc* to load or update histograms and N-tuples which should be analysed later. The implementation of the service depends only on the interfaces of the data service.

16

User Interface

The user interface is the component through which the end user interacts with the components of the system. In general, the components have interfaces to allow clients to configure and control them. The user interface component is the bridge between the internal interface (C++) and an interface suitable for human interaction.

16.1 Purpose and Functionality

The user interface (*UI*) is the more or less complex component through which the end user interacts with the rest of the system. One can immediately think of three ways in which this interaction could take place:

- A command line interpreter a la UNIX/DOS/VMS etc.
- A Graphical User Interface (GUI). For example one could imagine a tool not unlike explorer in order to look at the objects within the event data store (via the event data server of course).
- An interactive (as opposed to passive) event display. Here selecting objects via clicking is an operation which could act on objects, services or algorithms within the framework.

From the UI one would like to be able do at least the following:

- See and modify the properties of any component of the system (services, algorithms, etc.).
- Get the next event.
- Create some *data objects*.
- Make a selection of data objects from a *transient data store*.
- Create new *Algorithms*.
- Run an *Algorithm*.
- Make use of a service.
- From a command line *UI* instantiate a passive (i.e. non-interactive) event display.

16.2 Interfaces

There are two types of interface in the *UI*.

- The interface that the user interacts with. This is completely unspecified, and is completely under the control of the builder of any specific *UI*.
- The interface between the *UI* component and the rest of the framework. This is what must be specified concretely and must be adhered to exactly by all concrete UIs. In the case that the UI is the client of other components of the system, the *UI* will limit itself to just call the existing interfaces of the other components. The other case, where the client is one of the other components of the system is more complicated. For example if one component wants to alert the end user or affect in some way the appearance of the *UI* (change the color of a button, pop up a message, produce a beep, etc.) we need to define a standard way for the client component to interact with the *UI*. For that, we will need to specify some basic event[1] model.

1 Event is the sense of software event and not physics event.

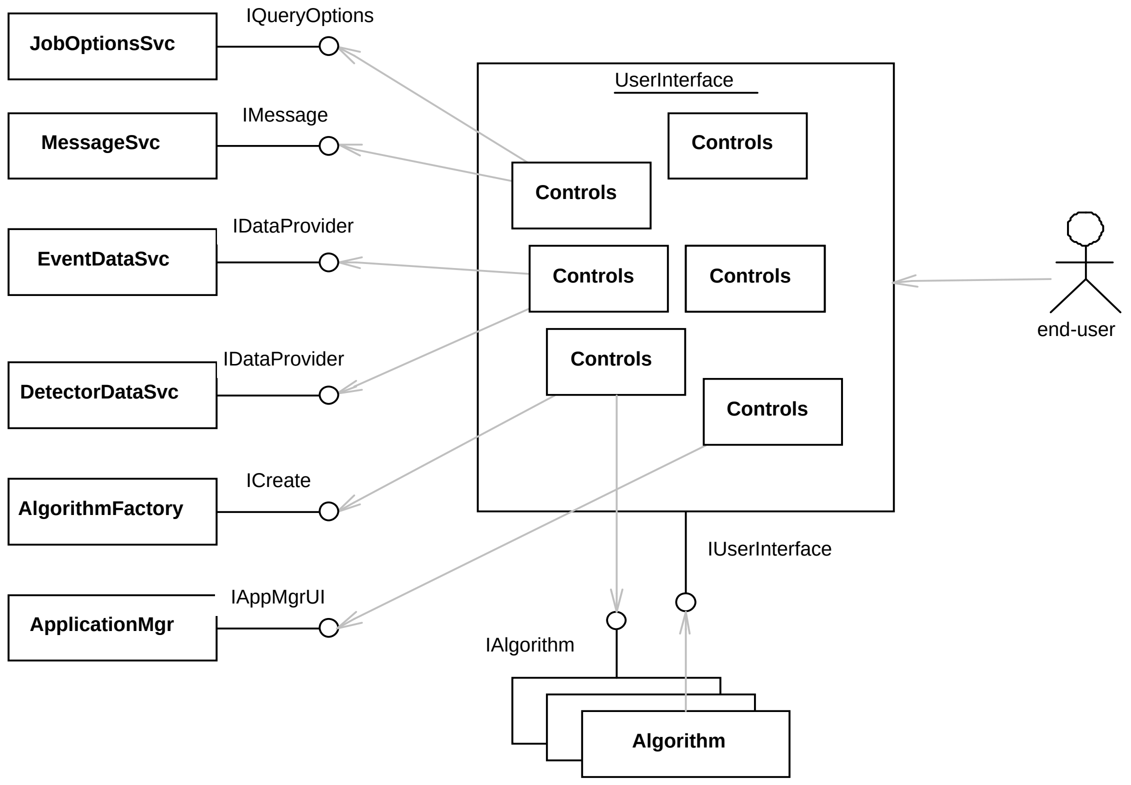


Illustration 16.1 The User Interface class diagram

17

Message Service

The message service publishes messages originating from other software components. It is the only component of the architecture that is allowed to transmit messages to the outside world. The service filters messages according to their severity and dispatches them to different output destinations.

17.1 Purpose and Functionality

The message service publishes the messages originating from other software components. It is the only component of the architecture that is allowed to transmit messages to the outside world. The service filters messages according to their severity and dispatches them to different output destinations (such as the process' standard output and standard error streams, an error logging facility, the detector control system etc.).

A given message may be generated several times. The message service can count messages of a given type; it may filter out duplicate occurrences of a given message.

The type of filtering and the selection of output destinations is configurable via job options.

In the initial implementation there will be only two possible output destinations (standard output and standard error)

17.2 Interfaces

The *MessageSvc* provides the following interfaces:

- **IMessage** interface. This interface is used by clients wishing to report a message to the *MessageSvc*
- **IMsgLogger** interface. This interface is used to dispatch messages to message logging clients. It will not be implemented in the initial version of the *MessageSvc*
- **IsetProperty** interface. The *MessageSvc* must implement this interface in order to get configured

17.3 Dependencies

- **Message severity levels** must be uniquely defined throughout the system
- **Message types** must be identified by a unique combination of facility code and error number
- **Message sources (and destinations)** must be uniquely identified.

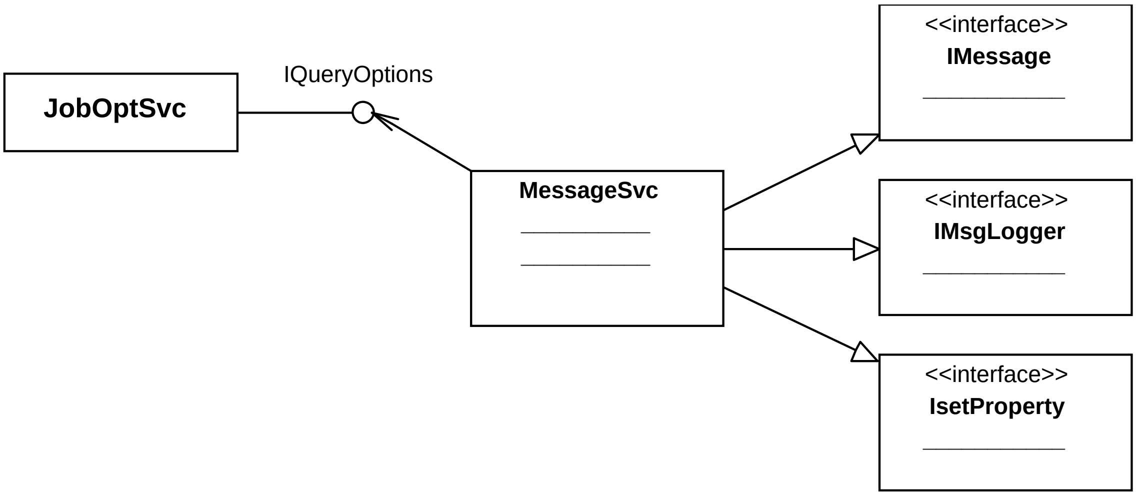


Illustration 17.1 IThe MessageSvc class diagram

18

Transient Event Data Model

In this chapter we introduce the LHCb specific data model we are considering for the framework.....

18.1 Purpose and Functionality

Transient event model (TEM) describes the representation of event data in the memory. The transient representation should be optimized for execution efficiency and code readability. TEM will be used in all LHCb data handling applications and should allow code reuse between them (simulation, reconstruction, analysis, event display, trigger level 2 and 3, etc.). The application code should be shielded from any kind of event data persistency.

TEM consists of class definitions and relationships between transient event objects. It contains the following class collections:

- **EventKernel** (contains supporting classes, e.g. TimeStamp, Classification, ProcessingVersion, RandomNumberSeed, ParticleID, CellId, TriggerPattern, etc.)
- **EventData** (identifiable classes, Event, MCEvent, RawEvent, ReconstructedEvent, AnalysisEvent, ObjectSet, and closely related classes Run, EventTag)
- **MonteCarloEvent** (the Monte Carlo truth information)
- **RawEvent** (output from DAQ or Monte Carlo)
- **ReconstructedEvent** (output of reconstruction program, not competed yet)
- **AnalysisEvent** (analysis objects, not completed yet)

Basic container objects which contain event entities (e.g. digits, tracks) are of type ObjectSet. This parametrized class implements one of the STL container classes vector<T> or list<T> (other STL container classes are not foreseen). As there is a need that the contained objects know which mother container they belong to, it is necessary to re-implement the STL-like interface in the class ObjectSet.

Objects of most of the types from the high level event structure have to be identifiable. It means, that they have to inherit from class DataObject, which holds the Identifier.

The current TEM has been developed according the requirements known from current LHCb simulation and analysis application (SICB). The complete SICB output can be stored using the current TEM.

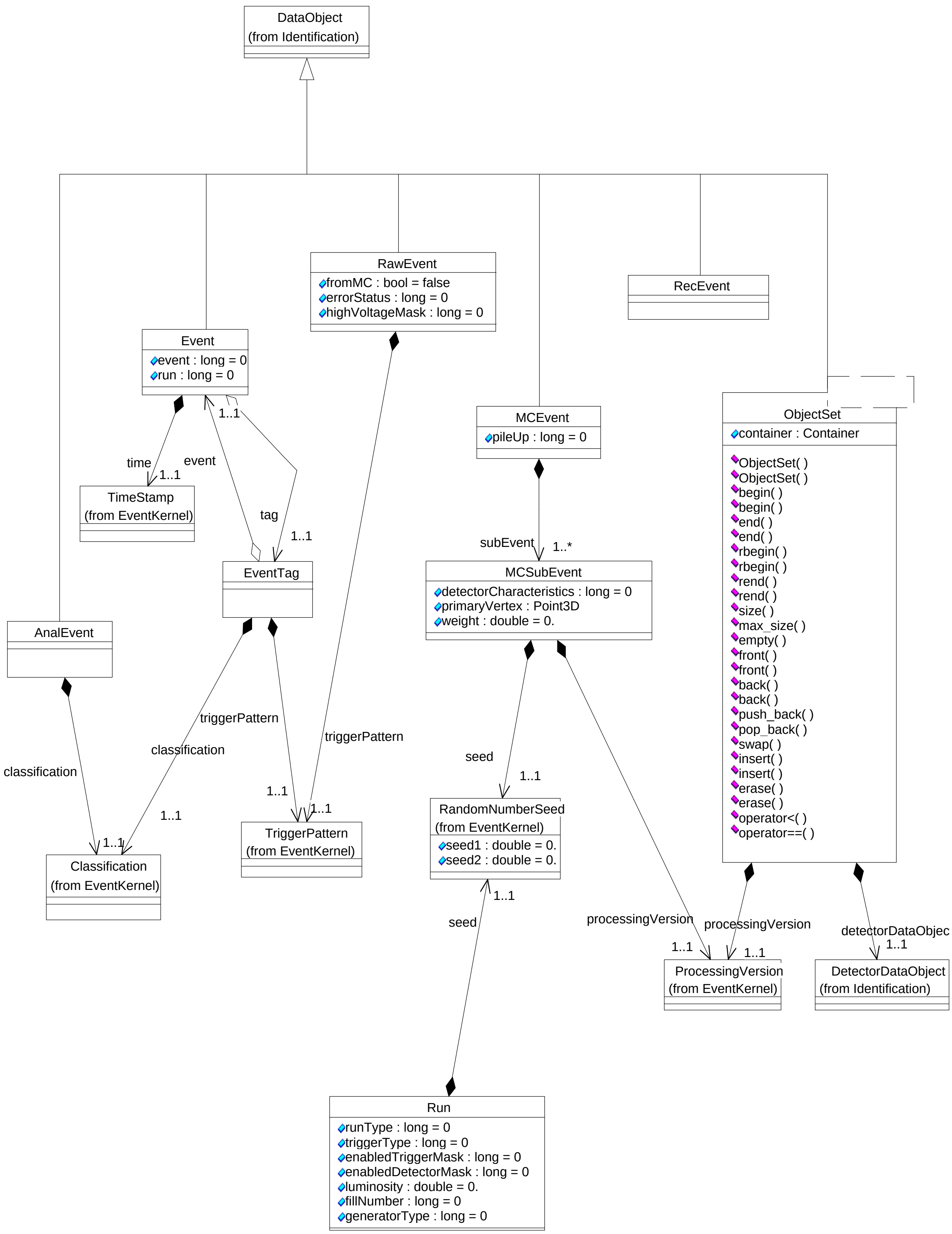
18.2 Access and Interfaces

Finding objects and registering newly created objects of TEM has always to go through the Event Data Service (EventDataSvc). Algorithms can request objects of TEM from the Event Data Service, which makes them available (either they are already in the transient data store, or it requests them to be loaded from Persistent Data Store, if available), i.e. it passes the reference of the objects back to the algorithm.

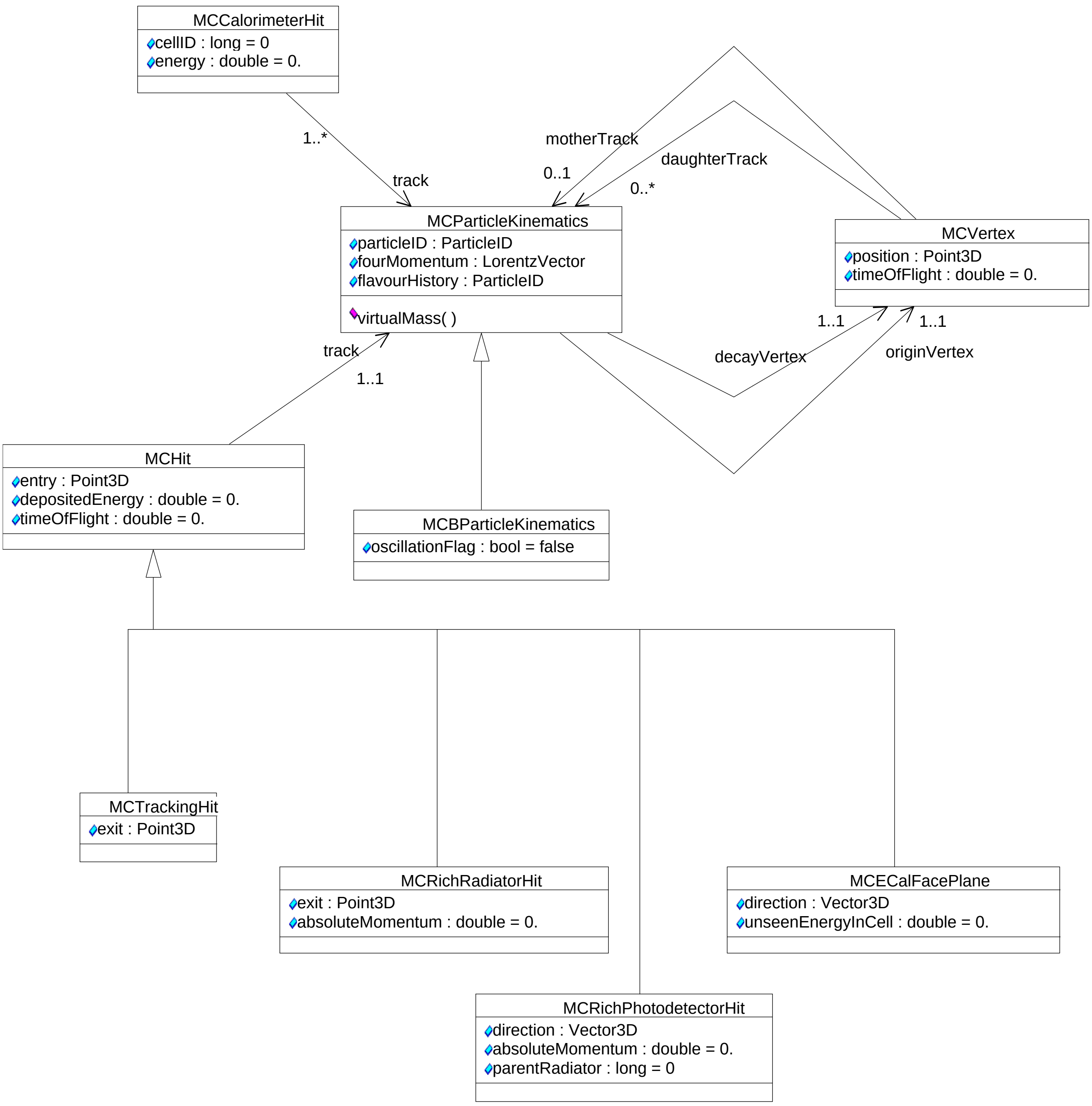
18.3 Dependencies

The TEM is the basic building unit, used by other LHCb components. It will depend on the identification package. The TEM uses heavily the Standard Template Library (STL) and Class Library for High Energy Physics (CLHEP).

18.4 EventData



18.5 MonteCarloEvent



18.6 RawEvent