anadrol

053.314.328-40 Identificação

custom\_dictionaries\_only

custom\_terms\_only

WeServeEveryone Clinic

1111 First Street California

111-111-11111 Fax: 111-111-1111 Chart Summary

Jane Latte

Home: 444-444-4444

Female DOB: 04/04/1950 0000-44444 Ins: Commercial xxxxx

Mr.

St.

Mrs.

Miss.

Patient Information

12345678

Name: Monica Latte Home Phone: 444-444-4444

Address: 4444 Coffee Ave

Chocolate, California Office Phone:

Patient ID: 0000-44444 Fax:

Birth Date: 04/04/1950 Status: Active

Gender: Female Marital Status: Divorced

Contact By: Phone Race: Black

Soc Sec No: 444-44-4444 Language: English

Resp Prov: Carl Savem MRN: MR-111-1111

Referred by: Emp. Status: Full-time

Email: Sens Chart: No

Home LOC:WeServeEveryone External ID: MR-111-1111

Problems

DIAEBETES MELLITUS (ICD-250.)

HYPERTENSION, BENIGN ESSENTIAL (ICD-401.1)

Code: V192.1

Medications

PRINIVIL TABS 20 MG (LISINOPRIL) 1 po qd

Last Refill: #30 x 2 : Carl Savem MD (08/27/2010)

HUMULIN INJ 70/30 (INSULIN REG & ISOPHANE (HUMAN)) 20 units ac breakfast

Last Refill: #600 u x 0 : Carl Savem MD (08/27/2010)

Directives

Allergies and Adverse Reactions (! = critical)

Services Due

FLU VAX, PNEUMOVAX, MICROALB URN

3/18/2011 - Office Visit: F/u Diaebetes

Provider: Carl Savem MD

Location of Care: WeServeEveryone Clinic

OtherSavices

new{}

new{}

initialize

getOptions

AlgorithmFactory

new{}

object

Algorithm

new{} initialize

getOptions

initialize

create

main() create

run

EventSelector

select

EventDataSvc

DetectorDataSvc

initialize

initialize

find

register

newevent

doevent

doevent

syncho

EvtPersistencySvc

load

save

DataItemSelector

save

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

System Design

30 GAUDI

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.

Component Interactions

GAUDI 31

Event Transient Store

Algorithm

object

Converter

EvtPersistencySvc

EventDataSvc

Persistent

StorageMS

1:retrieveObject

2:getAddress

Node:DataObject

:DataObject

3

4:createObj

5:createObj

6:read

7:new{}

8

9:register

10

EventDataSvc

ApplicationMgr

1:setRoot

2:retrieveObj

3:checkValidity

Detector Transient Store

DetectorDataSvc

:DataObject

4:checkValidity

DetPersistencySvc

5:updateObj

object

Converter

Persistent

6:updateObj StorageMS

7:read

8:update

(a)

(b)

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.

System Design

32 GAUDI

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.

Physical Design

GAUDI 33

A package x DependsOn another package y if 1 or more

components in x DependsOn one or more components in y

k l

i j

Package Level 2 Level 2

Level 1

f g

b

a h

c d e

Package Level 1

Package a

Package b

DependsOn

v

s

k l

u

r

p q

i g h o

a b c j t d m e f n

v

s

k l

u

r

p q

o

g h

i

a b c

j

t

d

m

e f

n

Pkg A

Pkg B Pkg C

Pkg D

L1

L2

L1

L2

L1

L2

L3

L4

L1

L2

L1

L2

L3

L4

L5

L6

Illustration 3.8 Components, packages and their dependencies

System Design

34 GAUDI

GAUDI 35

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 . . . . . . . . . . . . . . . . 36

4.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 36

4.3 Dependencies . . . . . . . . . . . . . . . . . . . . . . . 38

Application Manager

36 GAUDI

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.

Interfaces

GAUDI 37

ApplicationMgr

\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_

run()

<<interface>>

IProperty

\_\_\_\_\_\_\_\_\_\_

setProperty()

getProperty()

<<interface>>

ISvcLocator

\_\_\_\_\_\_\_\_\_\_

getService()

existsService()

addService()

removeService()

<<interface>>

IAppMgrUI

\_\_\_\_\_\_\_\_\_\_

nextEvent()

terminate()

finalize()

initialize()

MessageSvc

JobOptionsSvc

IQueryOptions

IMessage

EventDataSvc

AlgorithmFactory

ICreate

DetectorDataSvc

EventSelector

ISelector

ListSvc

ListAlg

IDataMgr

IDataProvider

IDataProvider

IDataMgr

Configured

Initialized

Active

ChangeCond DoingEvent

Stopped

(a)

(b)

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

Application Manager

38 GAUDI

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.

GAUDI 39

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 specification1. 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.

5.1 Purpose and Functionality . . . . . . . . . . . . . . . . 40

5.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 40

5.3 Dependencies . . . . . . . . . . . . . . . . . . . . . . . 40

1 Except ofcourse for specialised things such as converters.

Algorithms

40 GAUDI

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 Algorithms.

• Algorithms. The Algorithm may use other smaller algorithms to implement its function.

• Other services like EvtPersistecySvc, EventSelector, etc. are initialized by the ApplicationMgr in

Dependencies

GAUDI 41

Algorithm

\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_

<<interface>>

IProperty

\_\_\_\_\_\_\_\_\_\_

setProperty()

getProperty()

<<interface>>

IAlgorithm

\_\_\_\_\_\_\_\_\_\_

doChange()

execute()

finalize()

initialize()

MessageSvc

JobOptionsSvc

IQueryOptions

IMessage

EventDataSvc

AlgorithmFactory

ICreate

DetectorDataSvc

IDataProvider

IDataProvider

(a)

Initialized

Active

Offline

finalize() initialize()

execute()

doChange()

(b)

Illustration 5.1 (a) The Algorithm class diagram

(b) The Algorithm internal state diagram

Algorithms

42 GAUDI

GAUDI 43

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 . . . . . . . . . . . . . . . . 44

6.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 46

6.3 Dependencies . . . . . . . . . . . . . . . . . . . . . . . 46

Data Converter

44 GAUDI

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.

Purpose and Functionality

GAUDI 45

IInterface

<<Interface>>

IConverter

<<Interface>>

Converter

<<Implementation>

ObjtyConverter

<<Implementation>

ConcreteConverter

<<Implementation>

SicBxxConverter

<<Implementation>

IConverter

initialize () : StatusCode

getObjDataType () : const Class\*

getRepSvcType () : unsigned char

createObj (pAddress : IOpaqueAddress\*, refpObject : DataObject\*&) : StatusCode

fillObjRefs (pAddress : IOpaqueAddress\*, pObject : DataObject\*) : StatusCode

updateObj (pAddress : IOpaqueAddress\*, pObject : DataObject\*) : StatusCode

updateObjRefs (pObject : DataObject\*) : StatusCode

createRep (pObject : DataObject\*, refpAddress : IOpaqueAddress\*&) : StatusCode

fillRepRefs (pAddress : IOpaqueAddress\*, pObject : DataObject\*) : StatusCode

updateRep (pAddress : IOpaqueAddress\*, pObject : DataObject\*) : StatusCode

updateRepRefs (pAddress : IOpaqueAddress\*, pObject : DataObject\*) : StatusCode

<<Interface>>

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.

Data Converter

46 GAUDI

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.

GAUDI 47

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 . . . . . . . . . . . . . . . . 48

7.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 48

7.3 Dependencies . . . . . . . . . . . . . . . . . . . . . . . 50

Job Options Service

48 GAUDI

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.

Interfaces

GAUDI 49

JobOptSvc

\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_

<<interface>>

IQueryOptions

\_\_\_\_\_\_\_\_\_\_\_

configure()

setMyProperties()

getOption()

AClient

IsetProperty

OptionsArchive OptionsEditor

Illustration 7.1 The JobOptionsSvc class diagram

Job Options Service

50 GAUDI

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.

GAUDI 51

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 . . . . . . . . . . . . . . . . 52

8.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 52

Event Selector

52 GAUDI

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.

Interfaces

GAUDI 53

MessageSvc

JobOptionsSvc

IQueryOptions

IMessage

EventSelector

\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_

Selector

<<interface>>

IProperty

\_\_\_\_\_\_\_\_\_\_

setProperty()

getProperty()

<<interface>>

IEvtSelector

\_\_\_\_\_\_\_\_\_\_

setCriteria()

getIterator()

<<interface>>

ISelectorUI

\_\_\_\_\_\_\_\_\_\_

Illustration 8.1 The EventSelector class diagram

Event Selector

54 GAUDI

GAUDI 55

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 . . . . . . . . . . . . . . . . 56

9.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 56

9.3 Dependencies . . . . . . . . . . . . . . . . . . . . . . . 56

Transient Data Store

56 GAUDI

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).

Dependencies

GAUDI 57

Specific

Generic

Identifiable

Directory

<<interface>>

IClassInfo

DataObject

Event

RecEvent

EventTag

EcalHits EcalClusts

Hit Cluster

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

Transient Data Store

58 GAUDI

GAUDI 59

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 . . . . . . . . . . . . . . . . 60

10.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 60

10.3 Dependencies . . . . . . . . . . . . . . . . . . . . . . . 62

Event Data Service

60 GAUDI

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.

Interfaces

GAUDI 61

IDataManagerSvc

<<Interface>>

IDataProviderSvc

<<Interface>>

IService

<<Interface>>

Service

<<Implementation>>

EventDataSvc

<<Implementation>>

IEventDataSvc

<<Interface>>

Specific interface

not existant

- for future extensions

IInterface

<<Interface>>

\_root

0..1

IRegistryEntry

<<Interface>>

\_dataLoader

0..1

IConversionSvc

<<Interface>>

DataSvc

<<Implementation>>

Generic data service

Transient

Data Store

Service to retrieve

nonexistant objects

Availible

Interfaces

IDataManagerSvc

setRoot (pRootObj : DataObject\*, top\_path : const string&) : StatusCode

setDataLoader (service : IPersistencySvc\*) : StatusCode

clearStore () : StatusCode

clearSubTree (sub\_tree\_path : const string&) : StatusCode

clearSubTree (pObject : DataObject\*) : StatusCode

traverseTree (pSelector : ISelectionAgent\*) : StatusCode

traverseSubTree (sub\_tree\_path : const string&, pSelector : ISelectionAgent\*) : StatusCode

traverseSubTree (pObject : DataObject\*, pSelector : ISelectionAgent\*) : StatusCode

<<Interface>>

IDataProviderSvc

retrieveObject (full\_path : const string&, refpObject : DataObject\*&) : StatusCode

retrieveObject (root\_obj : DataObject\*, refpObject : DataObject\*&) : StatusCode

registerObject (root\_path : const string&, obj : DataObject\*) : StatusCode

registerObject (root\_obj : DataObject\*, obj : DataObject\*) : StatusCode

unregisterObject (path : const string&) : StatusCode

unregisterObject (obj : DataObject\*) : StatusCode

findObject (full\_path : const string&, pRefObject : DataObject\*&) : StatusCode

findObject (pNode : DataObject\*, leaf\_name : const string&, pRefObject : DataObject\*&) : StatusCode

<<Interface>>

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

Event Data Service

62 GAUDI

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.

GAUDI 63

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 . . . . . . . . . . . . . . . . 64

11.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 64

11.3 Dependencies . . . . . . . . . . . . . . . . . . . . . . . 66

Event Persistency Service

64 GAUDI

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.

Interfaces

GAUDI 65

IService

<<Interface>>

IInterface

<<Interface>>

Service

<<Implementation>

IPersistencySvc

<<Interface>>

\_converter

1..1

IConverter

<<Interface>>

1..1

\_type

Class

<<Implementation>>

ConversionSvc

<<Implementation>

\_knownConverters

0..\*

ListEntry

<<struct>>

For later

improved

functionality

PersistentSvc

<<Implementation>

Service

Dispatcher

Workers responsible

for different database

technologies

\_dataServices

0..\*

IConversionSvc

<<Interface>>

EvtPersistencySvc

<<Implementation>

Availible

Interfaces

Extension 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

<<Interface>>

Illustration 11.1 The EvtPersistencySvc and the provisional interface.

Event Persistency Service

66 GAUDI

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.

GAUDI 67

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 . . . . . . . . . . . . . . . . 68

12.2 Interfaces . . . . . . . . . . . . . . . . . . . . . . . . . 70

12.3 Dependencies . . . . . . . . . . . . . . . . . . . . . . . 70

Detector Data Service

68 GAUDI

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.

Application to Establish an Accessory Dwelling Unit or Backyard Cottage

City of Seattle

Department of Planning and Development

Application to Establish an Accessory Dwelling Unit

I am (check one)

Applying to create a new accessory dwelling unit.

Applying to legalize an existing unauthorized unit. As reflected in King County real estate

records, I purchased the lot on which the unauthorized unit is located less than one year ago,

and am submitting proof of this purchase. (No penalty)

Applying to voluntarily legalize an existing unauthorized unit. I may be subject to civil penalties until the permit process is completed.

Project Number:987525

Address: 60 Vine Street Noblesville, Indiana 46060

Owner: Jeffrey Hall

Daytime Phone # 154-087-7612

Assessor’s Parcel Number:1098745

Submit this form along with required plans and other documents.

Parking Waiver Request, if necessary

Not located within a Residential Parking Zone (RPZ)

Located within a RPZ; parking waiver study included

Phone:154-087-7612

Name(s) of Tenant(s)

Phone:154-933-0812

Phone:154-254-8934

Owner Occupancy Covenant, completed, notarized, recorded; original to DPD.

Date Unit was Created (to best of your knowledge):02/02/2016

Value of Construction Work Needed to Legalize Unit:

Copy of the Contractor’s Registration/Lien Law Form (completed)

Copy of Agent’s Authorization Letter from Owner (if agent)

Applicant’s Name: Kimberly Harris

Date received:09/03/2016

(PLEASE PRINT)

Applicant Signature

Date signed:09/03/2016

Relationship of applicant: (circle one) owner, agent, architect, contractor, engineer

Receipt #:94

Date of receipt: 10/03/2016

For DPD Use Only (must be completed)

Urban Village /

Urban Center

Application

Parking

Parking

Waiver Granted

# of Parking

Spaces Provided

Unit Square Footage

\_\_\_\_\_\_\_ Detached

(BYC)

No

No

No

\_\_\_\_\_\_\_\_\_\_\_ space

\_\_\_\_\_\_\_\_\_\_\_ space

\_\_\_\_\_\_\_ Attached

(ADU)

Page 1 of 2

?Application to Establish an Accessory Dwelling Unit or Backyard Cottage

City of Seattle

Department of Planning and Development

Application to Establish a Backyard Cottage

I am (check one):

Applying to create a new accessory dwelling unit.

Applying to legalize an existing unauthorized unit. As reflected in King County real estate

records, I purchased the lot on which the unauthorized unit is located less than one year ago,

and am submitting proof of this purchase. (No penalty)

Applying to voluntarily legalize an existing unauthorized unit. I may be subject to civil penalties until the permit process is completed.

Project Number

Address

Owner

Daytime Phone #165-892-0982

Assessor’s Parcel Number

Submit this form along with required plans and other documents.

Parking Waiver Request, if necessary

Not located within a Residential Parking Zone (RPZ)

Located within a RPZ; parking waiver study included

Phone:

Name(s) of Tenant(s):Alice Long

Phone:

Phone:

Owner Occupancy Covenant, completed, notarized, recorded; original to DPD.

Date Unit was Created (to best of your knowledge):

Value of Construction Work Needed to Legalize Unit:

Copy of the Contractor’s Registration/Lien Law Form (completed)

Copy of Agent’s Authorization Letter from Owner (if agent)

Applicant’s Name

Date received

(PLEASE PRINT)

Applicant Signature

Date signed

Relationship of applicant: (circle one) owner, agent, architect, contractor, engineer

Receipt #

Date of receipt

For DPD Use Only (must be completed for units in single family zones)

Urban Village /

Urban Center

Application

Parking

Parking

Waiver Granted

# of Parking

Spaces Provided

Unit Square Footage

\_\_\_\_\_\_\_ Detached

(BYC)

Yes \_\_ No \_\_

Yes \_\_ No \_\_

Yes \_\_ No \_\_

\_\_\_\_\_\_\_\_\_\_\_ space

\_\_\_\_\_\_\_\_\_\_\_ space

\_\_\_\_\_\_\_ Attached

(ADU)

Page 2 of 2

?