# *GiGa*

***G**eant4 **I**nterface for **G**audi **A**pplication*
*or*
***G**audi **I**nterface for **G**eant4 **A**pplication*

Version 4.0 (March'**2k+1**)

Ivan Belyaev[1]
(*ITEP, Moscow*)

[1]`E-mail: Ivan.Belyaev@itep.ru`

## Abstract

The way to access for GEANT4 facilities from GAUDI framework is described. The two-layered structure of `GiGaSvc` and GIGA `Conversion Service`s for communications of `Algorithm`s within GAUDI framework with GEANT4 structures allows the usage of GEANT4 tool kit as a black-box without detailed knowledge of its internal features.

## Table of Contents

# Chapter 1: General

## 1.1 Simulation environment

Simulation is an essential part of the general software in modern high energy physics. It is foreseen to use **GEANT4** Tool Kit as the major simulation package for the LHC era.

### 1.1.1 Natural decomposition of simulation environment

It is worth to consider any simulation program as sequence of steps, each of them could be considered as as compact unit with some well defined input and output data flows.

For almost each simulation program it is natural to identify the following steps:

- Initialisation:
  At this step simulation program requires to be provided with physical properties of all participating components, like particles properties, properties of physical processes, description of geometry and materials.
- Event loop
  - Event initialisation:
    At this step simulation program requires to be provided with some event input data - initial kinematics
  - Event processing:
    At this step simulation program usually neither require to be provided with some input data nor produce some output data, but some user action methods are to be supplied to the simulation program (e.g. **`gustep.F`** routine in **GEANT3** package)
  - Event finalisation:
    At this step simulation program is ready to provide the user program with the simulation output data - hits, digits and output kinematics - secondary particles.
- Finalisation

## 1.2 Communication categories

From above sketched rough scheme one could deduce that all communications of user program with simulation environment could be naturally subdivided into 2 categories:

- Callback category:
  Communications from this category are characterised by dealing mainly with internal structures of simulation program. They do not produce or consume the data from the outside of the simulation environment. Usually they overwrite some internal default methods

from simulation environment and thus changing the default behavoiur
Routine **`gustep.F`** from **GEANT3** package could be considered as a typical representative of such category.

- Input-output category:
  Communications from this category are characterised by either producing the data, which to be used outside of the simulation environment, like hits, digits, secondary particles, histograms and n-tuples, or they load data flow from the outside of the simulation program, like particle properties, properties of physical processes and detector description. Often communications from this category could be easily identified as input streams or output streams.
  Routine **`guout.F`** from **GEANT3** package could be considered as a typical representative of output stream category and **`gukine.F`** could be considered as an example of input stream category.

Also communications could be classified into 2 classes:

- Configuration communications:
  Such type of communication is used for configuration of the simulation environment and supplying it with input data which are constant for a some period and sometimes for the whole job lifetime.

- Event-by-event communications:
  Such type of communication is used on event-by-event basis.

## 1.3 GEANT4 essential

A schematic view of communications with **GEANT4** tool kit is presented on figure 1.1. Arrows represent the direction of data flows.

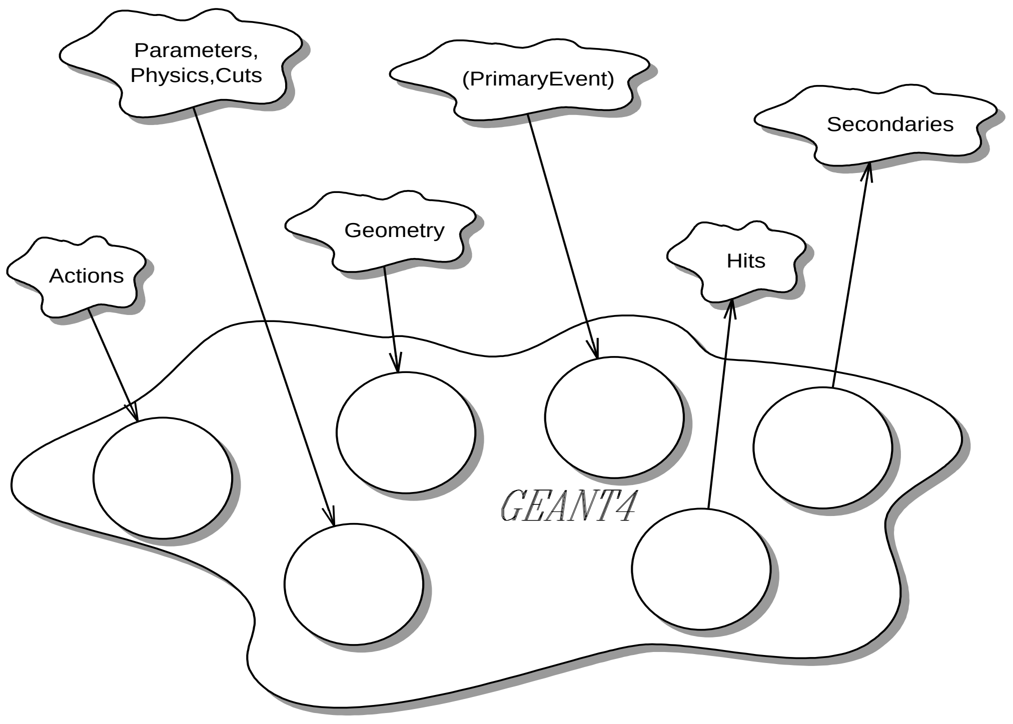

Figure 1.1: A schematic view of communications with **GEANT4** tool kit. Data flow directions are indicated by arrows.

Standard **GEANT4** based program requires to be configured. The configuration could be represented as communications between simulation environments, namely **`G4RunManager`** class and user program by set of user classes. User environment creates these classes, and simulation environment just uses them. The sketch is presented on figure 1.2.

**`G4RunManager`** class requires to be provided with 3 mandatory classes, implementing following interfaces

- **`G4VUserDetectorConstruction`**
  Concrete class, implementing this interface is responsible for creation of **GEANT4** geometry tree and material description.

- **`G4VUserPhysicsList`** Creation of particles, physics processes and tracking cuts are under responsibility of concrete class, implementing this interface.

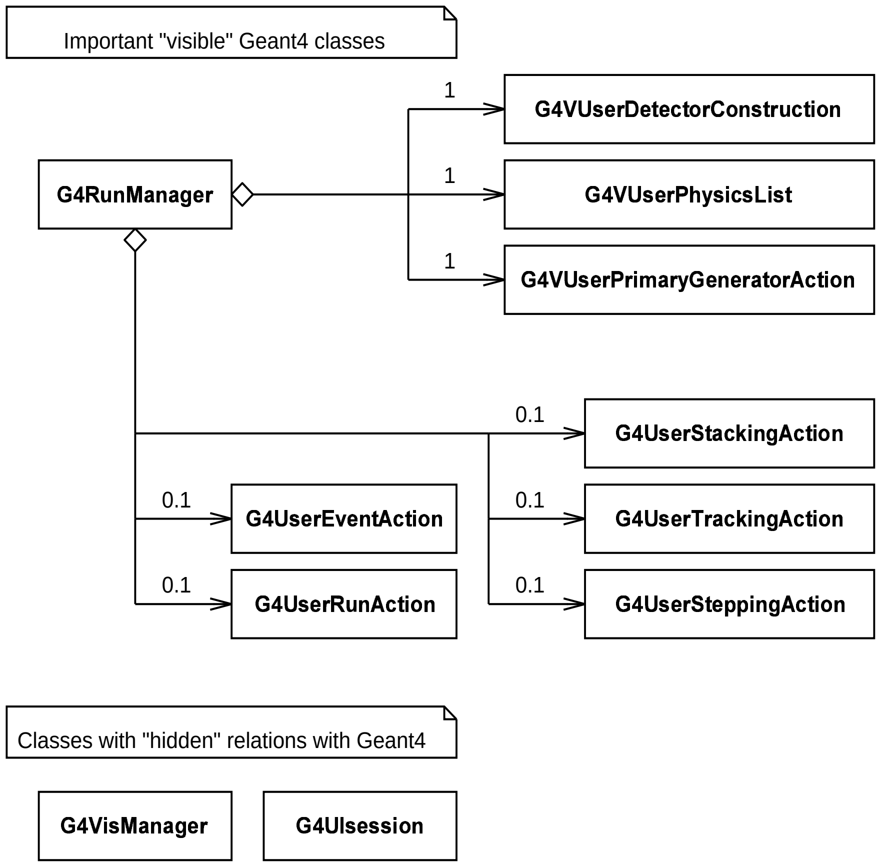

Figure 1.2: Sketch of classes, essential for configuration of **GEANT4** tool kit

- **`G4VUserPrimaryGeneratorAction`** The definition of kinematics of primary particles is under the control os concrete class, implementing this interface.

In addition **`G4RunManager`** could be provided with:

- **`G4UserRunAction`**,
  which defines specific callbacks to be executed in the begin and at teh end of each run.

- **`G4UserEventAction`** which defines specific callbacks to be executed in the begin and at the end of processiong of each event.

- **`G4UserStackingAction`**, which defined specific action for stacking method

- **`G4UserSteppingAction`**, which defines specific action to be performed at every step.

- **`G4UserTrackingAction`**, which defines specific action to be performed for each track.

The exist also 2 classes, which could be created independently and which communicate with **`G4RunManager`** in a quite hidden way:

- **`G4VisManager`** for visualisation
- **`G4UIsession`** for interactivity

### 1.3.1 Encapsulation of GEANT4 within GAUDI

To integrate **GEANT4** into **GAUDI** and to isolate **GEANT4** from end-user codes one need to put some efforts to perform following steps:

- Define the only one entry point for all communications with **GEANT4**.
  This is achieved via abstract interfaces **`IGiGaSvc`** and **`IGiGaSetUpSvc`**, which are implemented by **`GiGaSvc`**
- For all input/output streams one need to design the appropriate **`Conversion Service`**s and for all objects from input/output streams one need to implement proper **`Converter`**s
- Several mandatory technical actions also must be performed:
  - Provide the access to internal **GEANT4** event loop. This is achieved with implementation of **`GiGaRunManager`**
  - Allow the primary event to be constructed outside the **`G4VUserPrimaryGeneratorAction`** class. This goal is achieved with the proper collaboration between **`GiGaRunManager`** and **`GiGaSvc`**.
  - Allow the geometry tree to be constructed outside the **`G4VUserDetectorConstruction`** class. This is achieved with the proper collaboration between **`GiGaRunManager`** and **`GiGaSvc`**.

The rest of this document is devoted to a detailed description of this program step by step.

# Chapter 2: GiGaSvc Service

## 2.1 Why Service?

The most suitable form for embedding the simulation environment into GAUDI framework is **Service**. Several simple and independent GAUDI **Algorithm**s, **Converter**s and **Conversion Service**s could communicate with with **Service** suppling it with input data - detector description, particle properties, description of physical processes, cut-offs and initial kinematics and retrieving from it the output data in the format of hits and secondaries. At the top level, the **Service** is to be triggered by some algorithm.

The whole orchester of **Service**, its trigger **Algorithm** and its helper **Algorithm**s which are resposible for providing the **Service** with the input data and extraction the output data performs the perfect isolation of end-user codes from underlying simulation package.

The **Service** is devoted to be **the only one entry point** to simulation package. Unfortunately GEANT4 provides users with a lot of **static** accessors to it's key classes and therefore any direct communications with GEANT4 environment could not be prevented in a safe way. But any user **must avoid** any direct communications with GEANT4 environments not throught simulation **Service**.

Simulation **Service** implements 2 abstract interfaces:

- **IGiGaSvc** interface:
  for event-by-event communications
- **IGiGaSetUpSvc** interface:
  for configuration communications

## 2.2 IGiGaSvc interface

All event-by-event stream-like communications with simulation environment defined in **IGiGaSvc** abstract interface. This interface is designed to manipulate with input event data and output event data:

- Input event data are accepted in the form of
  - **G4PrimaryVertex***
    representing the GEANT4 primary event record
- Output event data, retrieved from the **Service** could be of the format:
  - **G4Event***
    provide the access to the whole internal processed event
  - **G4HCofThisEvent***
    provide the access to the all hit collections of the processed event

- **IGiGaSvc::CollectionPair***
  provide the access to the specific hit collection
- **G4TrajectoryContainer***
  provide the access kinematic of secondaries

Since all communications with simulation environment via **IGiGaSvc** interface are stream-like communications, the overloaded stream operators are considered as main methods in the interface definition:

```
class IGiGaSvc : virtual public IService
{
public:
  ...
  /// input  data
  virtual IGiGaSvc& operator << ( G4PrimaryVertex        * vertex       ) = 0 ;
  /// output data
  virtual IGiGaSvc& operator >> ( const G4Event*         & event        ) = 0 ;
  virtual IGiGaSvc& operator >> ( G4HCofThisEvent*       & collections  ) = 0 ;
  virtual IGiGaSvc& operator >> ( CollectionPair         & collection   ) = 0 ;
  virtual IGiGaSvc& operator >> ( G4TrajectoryContainer* & trajectories ) = 0 ;
  ...
};
```

In addition to these straightforward definitions, function-like methods are defined:

```
class IGiGaSvc : virtual public IService
{
public:
  ...
  /// input data
  virtual StatusCode  addPrimaryKinematics  ( G4PrimaryVertex  *        ) = 0 ;
  /// output data
  virtual StatusCode retrieveEvent          ( const G4Event*         & ) = 0 ;
  virtual StatusCode retrieveHitCollections ( G4HCofThisEvent*       & ) = 0 ;
  virtual StatusCode retrieveHitCollection  ( CollectionPair         & ) = 0 ;
  virtual StatusCode retrieveTrajectories   ( G4TrajectoryContainer*  & ) = 0 ;
  ...
};
```

## 2.3 IGiGaSetUpSvc interface

All configuration communications with simulation **Service** are defined in **GiGaSetUpSvc** interface.
The interface consists of the following definitions of operator-like calls:

```
class IGiGaSetUpSvc : virtual public IService
{
public:
  ...
  virtual IGiGaSetUpSvc& operator << ( G4VUserDetectorConstruction   * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4VPhysicalVolume             * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4VUserPrimaryGeneratorAction * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4VUserPhysicsList            * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4UserRunAction               * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4UserEventAction             * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4UserStackingAction          * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4UserTrackingAction          * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4UserSteppingAction          * ) = 0 ;
  virtual IGiGaSetUpSvc& operator << ( G4VisManager                  * ) = 0 ;
  ...
};
```

Here the advantage of operator-like methods is not so clear and obvious and ordinary function-like methods are defined in addition:

```
class IGiGaSetUpSvc : virtual public IService
{
public:
  ...
  virtual StatusCode setConstruction ( G4VUserDetectorConstruction   * ) = 0 ;
  virtual StatusCode setDetector     ( G4VPhysicalVolume             * ) = 0 ;
  virtual StatusCode setGenerator    ( G4VUserPrimaryGeneratorAction * ) = 0 ;
  virtual StatusCode setPhysics      ( G4VUserPhysicsList            * ) = 0 ;
  virtual StatusCode setRunAction    ( G4UserRunAction               * ) = 0 ;
  virtual StatusCode setEvtAction    ( G4UserEventAction             * ) = 0 ;
  virtual StatusCode setStacking     ( G4UserStackingAction          * ) = 0 ;
  virtual StatusCode setTracking     ( G4UserTrackingAction          * ) = 0 ;
  virtual StatusCode setStepping     ( G4UserSteppingAction          * ) = 0 ;
  virtual StatusCode setVisManager   ( G4VisManager                  * ) = 0 ;
  ...
};
```

## 2.4 GiGaSvc Service

Both **IGiGaSvc** and **IGiGaSetUpSvc** are implemented by a concrete class **GiGaSvc**. The class diagrams for **GiGaSvc** are shown on figures 2.1 and 2.2.

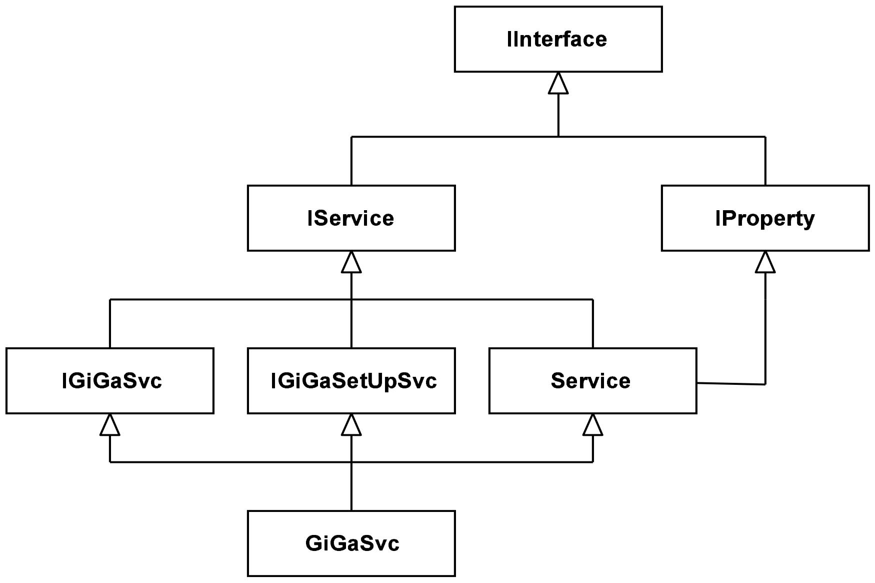

Figure 2.1: The inheritance diagram for **GiGaSvc** class

The **Service** creates and uses the concrete instance of **G4RunManager** class for managing of all GEANT4 classes. Since there could be only one object of type **G4RunManager** at the same type, only one instance of **GiGaSvc Service** could be instantiated.

The **GiGaSvc** creates a concrete instance of class **GiGaVisManager** for using GEANT4 visualization facilities. The creation of **GiGaVisManager** class is controled with boolean property **"UseVisManager"** of **GiGaSvc** class. The default value of this property is **false**.

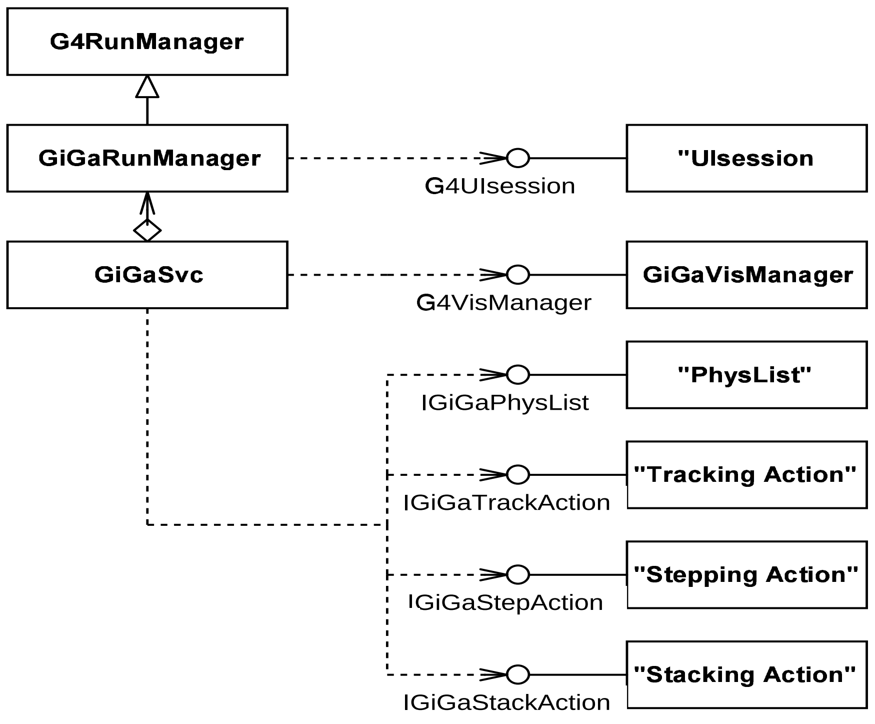

Figure 2.2: The simplified class diagram for **GiGaSvc** class. Some named components, located and used by service are not shown

The **GiGaSvc** takes the control on the instantiation (using abstract factories technique) of the objects of following types:

- **IGiGaPhysList**
- **IGiGaTrackAction**
- **IGiGaStepAction**
- **IGiGaStackAction**
- **IGiGaEventAction**

These configuration classes and interfaces are described in detail in chapter 4. The creation of these objects is under the control of corresponding properties of **GiGaSvc**:

- **"PhysicsList"**
- **"TrackingAction"**
- **"SteppingAction"**

- **"StackingAction"**
- **"EventAction"**

Each of these properties looks like: **"<ConcreteTypeName>/<InstanceNick>"**, similar to instantiation of **Algorithm**s and **Service**s. Deletion of these objects is performed by **GEANT4** kernel. **GEANT4** callback class **G4UserRunAction** is not recommended for usage within **GIGA** since one could not achieve proper sinchronization especially for termination of the last **G4Run**.

Essentially, that's all that **GiGaSvc** does and that's for what it was designed and implemented. All other it's functionality comes from delegation to **GiGaRunManager** class, which performs the real job.

All properties of **GiGaSvc** are listed in table 2.1.

Table 2.1: Properties of **GiGaSvc** and their default values.

| Property Name | Default Value |
|---|---|
| **"StartUIcommands"** | empty |
| **"EndUIcommands"** | empty |
| **"StartOfRunUIcommands"** | empty |
| **"EndOfRunUIcommands"** | empty |
| **"StartOfEventUIcommands"** | empty |
| **"EndOfEventUIcommands"** | empty |
| **"UIsessions"** | empty |
| **"ObjectManager"** | **"ApplicationMgr"** |
| **"PhysicsList"** | **""** |
| **"StackingAction"** | **""** |
| **"TrackingAction"** | **""** |
| **"SteppingAction"** | **""** |
| **"EventAction"** | **""** |
| **"UseVisManager"** | **false** |

## 2.5 GiGaRunManager

**GiGaRunManager** class is just a specialisation of general **G4RunManager** class. It inherits from it in a **private** way to restrict interface a little bit. Unfortunately due to static accessor to the base class the job could not be done in 100% safe way. as it was already mentioned before, users are not allowed to directly interact even with this class. All interaction should be done only via abstract interfaces of **GiGaSvc**class.

The essential public inteface consists of following declaration methods:

```
class GiGaRunManager: private G4RunManager
{
  friend class GiGaSvc;
  public:
  ...
  /// declarations
  virtual StatusCode declare( G4VUserPrimaryGeneratorAction  * ) ;
  virtual StatusCode declare( G4VPhysicalVolume              * ) ;
  virtual StatusCode declare( G4VUserDetectorConstruction    * ) ;
  virtual StatusCode declare( G4VUserPhysicsList             * ) ;
  virtual StatusCode declare( G4UserRunAction                * ) ;
  virtual StatusCode declare( G4UserEventAction              * ) ;
  virtual StatusCode declare( G4UserStackingAction           * ) ;
  virtual StatusCode declare( G4UserSteppingAction           * ) ;
  virtual StatusCode declare( G4UserTrackingAction           * ) ;
  virtual StatusCode declare( G4VisManager                   * ) ;
  ...
};
```

We need to get an easy access to **GEANT4**internal event loop and take a full control over event loop. This is achieved with implementation of three major methods, which perform the actual access to control the event-by-event processing:

```
class GiGaRunManager: private G4RunManager
{

  friend class GiGaSvc;
  public:
  ...
  virtual StatusCode  prepareTheEvent ( G4PrimaryVertex    * vertex = 0 ) ;
  virtual StatusCode  processTheEvent (                                 ) ;
  virtual StatusCode  retrieveTheEvent( const G4Event      *& event     ) ;
  ...
};
```

The following functions from **`G4RunManager`** is overwritten:

```
class GiGaRunManager: private G4RunManager
{
  protected:
  ...
  /// "the main" method of G4RunManager
  virtual  void BeamOn( int         n_event       ,
                        const char* macroFile =  0 ,
                        int         n_select  = -1 );
  ///
  void       InitializeGeometry() ;
  void       Initialize()         ;
  ...
};
```

The **`G4RunManager::beamOn(...)`** method should not be called. It is overwritten and disabled.

## 2.5.1 `GiGaRunManager` states

The state of **`GiGaRunManager`** is defined by set of flags:

- ”**GEANT4** kernel is initialized”
- ”**GEANT4** run is initialized”
- ”**GEANT4** event is prepared”
- ”**GEANT4** event is processed”

Invocation of any declaration method from **IGiGaSetUpSvc** interface automatically switched off all flags and reset **GiGaRunManager** to it's initial state. If declaration method is invoked when states "**GEANT4** event is prepared" or "**GEANT4** event is processed" is active, it automatically triggers the termination of current **G4Run**.
Current **GiGaRunManager** state could be inspected using following methods form it's public interface

```
class GiGaRunManager: private G4RunManager
{
  public:
  ...
  /// states
  inline bool krn_Is_Initialized () const ;
  inline bool run_Is_Initialized () const ;
  inline bool evt_Is_Prepared    () const ;
  inline bool evt_Is_Processed   () const ;
  ...
};
```

### 2.5.2 Three major methods of GiGaRunManager

Simplified interaction diagrams for 3 major methods of **GiGaRunManager** is shown in figures 2.3, 2.4 and 2.5.

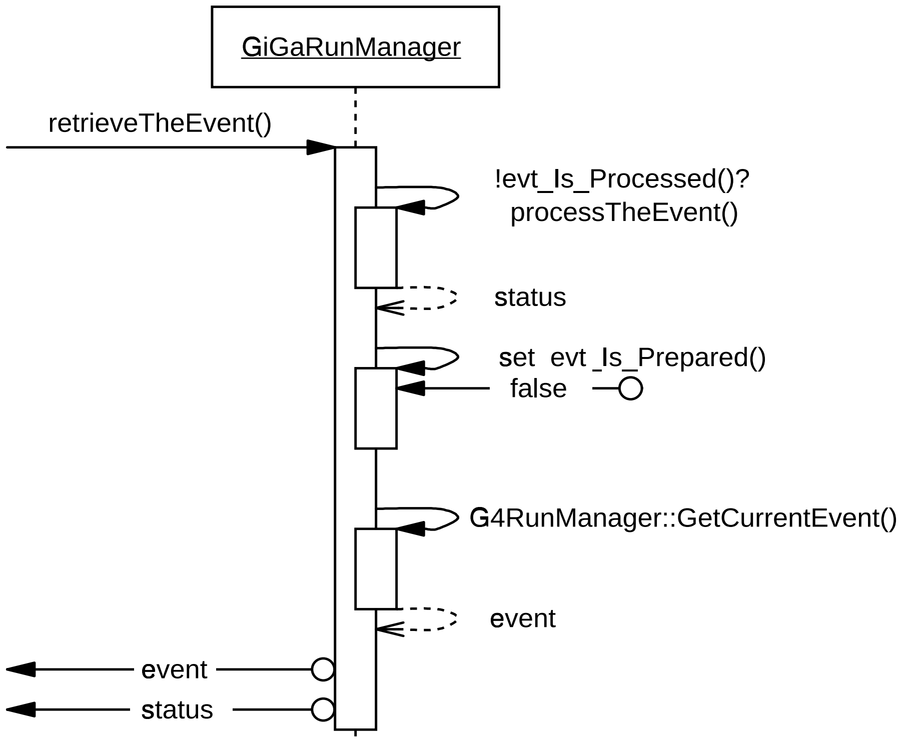

Figure 2.3: The simplified interaction diagram for **retrieveTheEvent(G4Event*&)** method of **GiGaRunManager**.

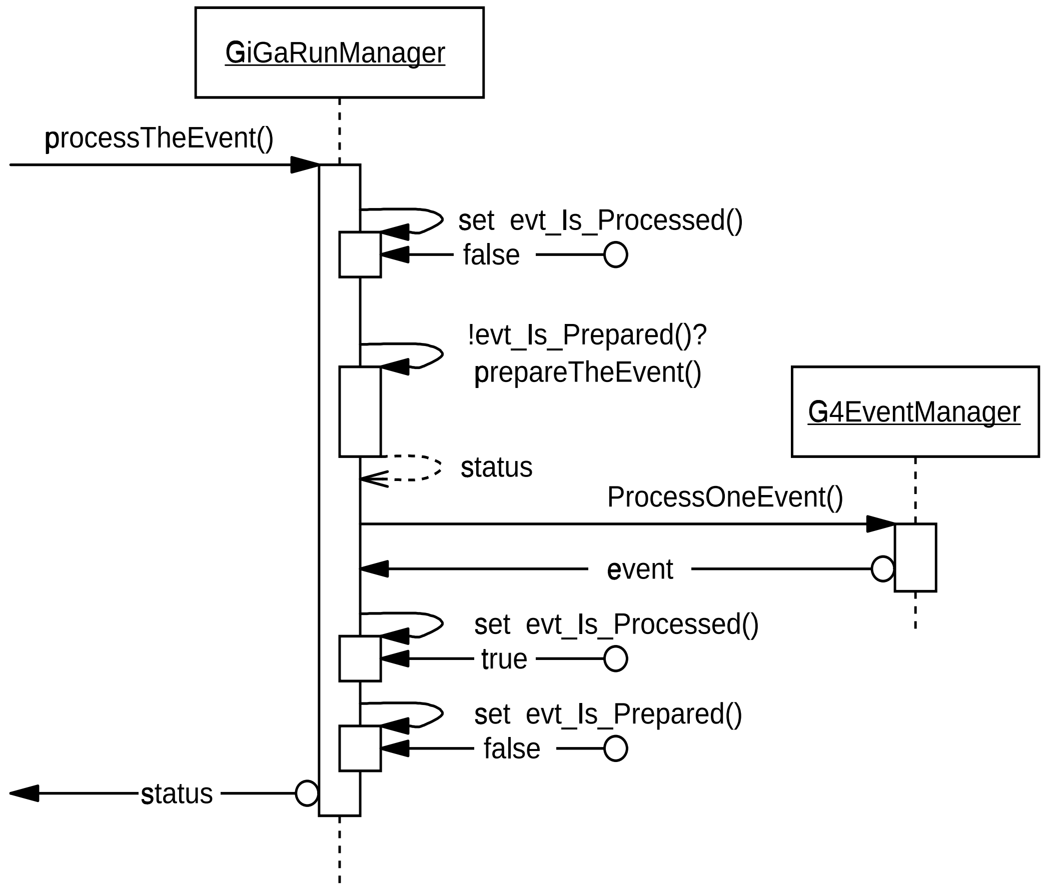

Figure 2.4: The simplified interaction diagram for **`processTheEvent()`** method of **`GiGa-RunManager`**.

One should mention here that **`prepareTheEvent(G4PrimaryVertex*)`** method does not only invoke **`G4Event::AddPrimaryVertex()`** method, but also takes care about creation of current event and deletion (or stacking it into special stack for ”previous events” - whcih could be useful for spill-over simulation) of the current event if is is already ”processed”

This method could be invoked repetedly to add several primary vertices into current event, thus constructing quite complicated primary event. It is a way how the pile-up could be implemented in a trivial and transparent way.

An additional feature of this method is that usage of this method does not forbid the usage of **`G4VUserPrimaryGeneratorAction*`** object for generation of the primary event! This option is triggered by invocation **`prepareTheEvent(0)`**; and one could combine both possibilities and use them separately or in conjunction for construction of quite non-trivial sophisticated kinematics of primary event.

All these features are not shown on simplified interaction diagram in figure 2.5.

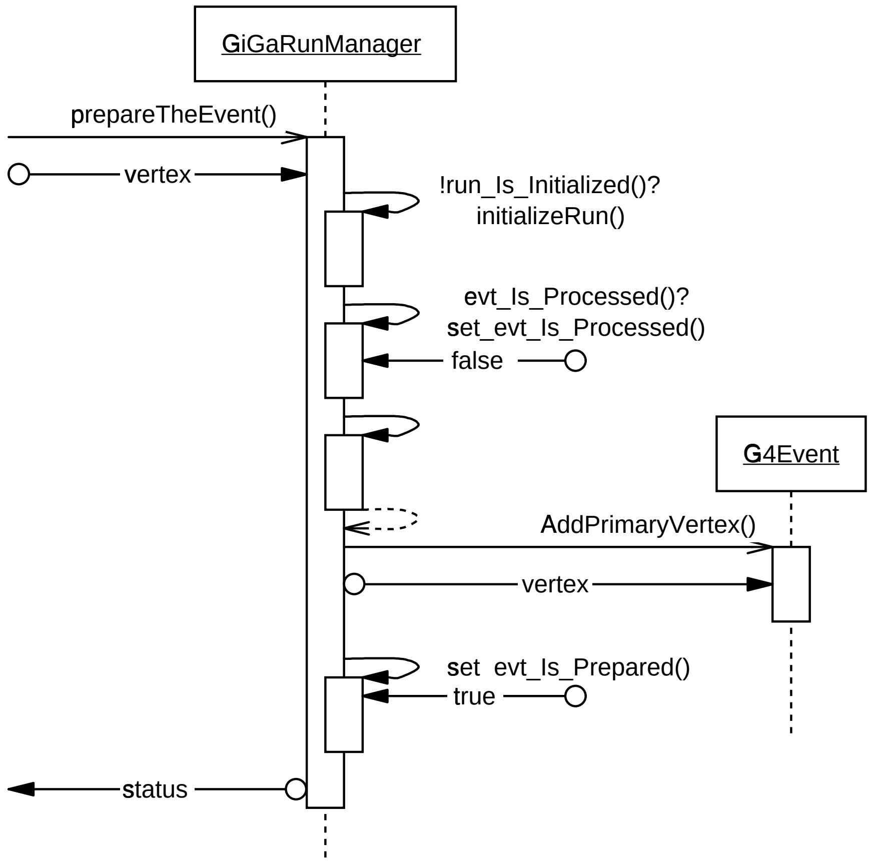

Figure 2.5: The simplified interaction diagram for **`prepareTheEvent()`** method of **`GiGaRunManager`**. Some important features are not shown, but explained in text.

### 2.5.3 Simulation ”on demand”

From simplified interaction diagrams which are shown in figures 2.3,2.4 and 2.5 one could easily deduce that **`GiGaRunManager`** allows to implement ”simulation on demand” approach. One just need to retrieve the output event from **`GiGaRunManager`** and it triggers itself to the sequence of self-initialization, event preparation and event processing, if event is not yet ready for output.

It is not clear wether one needs to have such option, but in current version of **GIGA** he have it for free.

### 2.5.4 Interaction with `GiGaSvc`

Simplified interaction diagram for communications between **`GiGaSvc`** and **`GiGaRunManager`** classes are shown on figure 2.6.

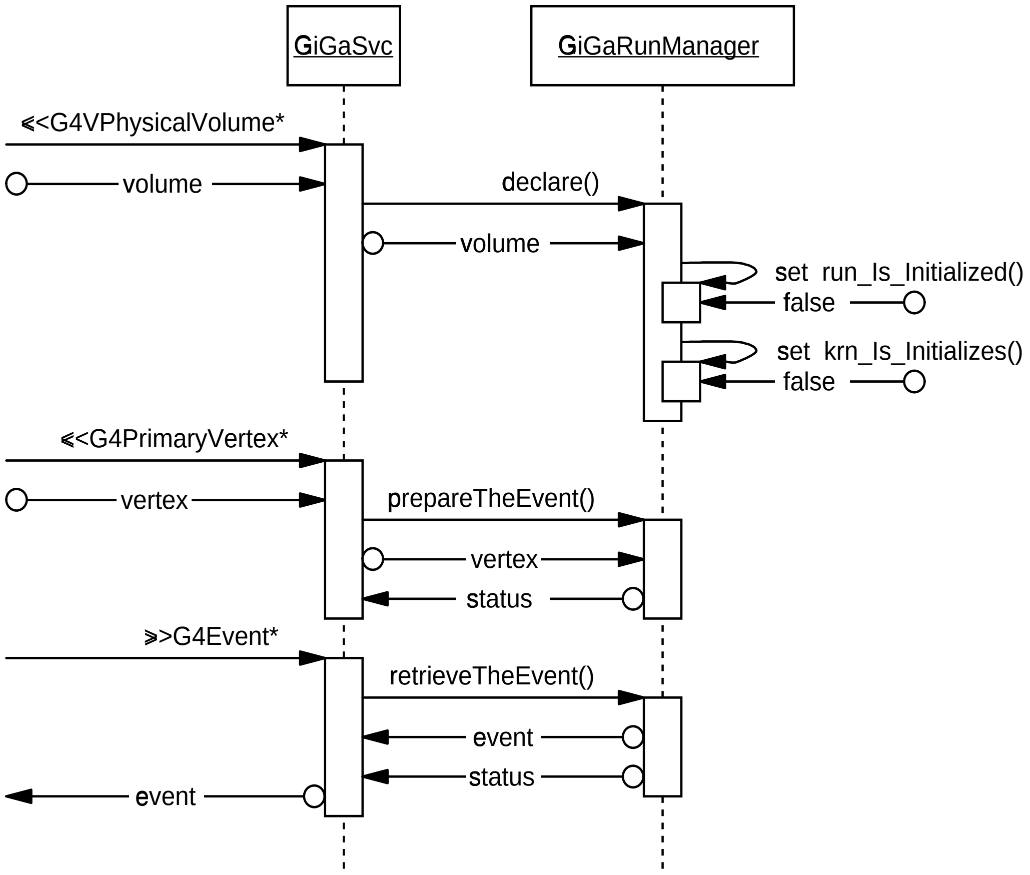

Figure 2.6: The simplified interaction diagram for **GiGaSvc** and **GiGaRunManager** classes

# Chapter 3: Conversion from/to GEANT4

## 3.1 GIGA **`Conversion Service`**s and **`Converter`**s

Conversion of **GAUDI** objects from and to **GEANT4** objects is under the control of set of **GIGA** **`Conversion Service`**s. The class diagrams for all **GIGA** **`Conversion Service`**s are shown in figure 3.1.

To simplify the configuration of each concrete service all concrete **`Conversion Service`**s inherits from base **`GiGaCnvSvcBase`** class, which takes care about all common components needed for **GIGA** **`Conversion Service`**s. This base class has following properties, listed in table 3.1. All these properties could be overwritten at run time.

Table 3.1: Properties of **`GiGaCnvSvcBase`** and their default values.

| Property Name | Default Value |
|---|---|
| **`GiGaService`** | **`"GiGaSvc"`** |
| **`GiGaSetUpService`** | **`"GiGaSvc"`** |
| **`EventDataProviderService`** | **`"EventDataSvc"`** |
| **`DetectorDataProviderService`** | **`"DetectorDataSvc"`** |
| **`ParticlePropertyService`** | **`"ParticlePropertySvc"`** |
| **`MagneticFieldService`** | **`"MagneticFieldSvc"`** |
| **`ChronoStatService`** | **`"ChronoStatSvc"`** |
| **`ObjectManager`** | **`"ApplicationMgr"`** |
| **`IncidentService`** | **`"IncidentSvc"`** |

The analogous base class **`GiGaCnvBase`** is devoted to be an common base for all **GIGA** **`Converter`**s. In addition to standard methods from **`Converter`** it implements several useful accessors:

```
class GiGaCnvBase: public Converter
{
  protected:
  /// useful acessors
  inline IGiGaCnvSvc*          cnvSvc    () const ; // "own" conversion service
  inline IGiGaGeomCnvSvc*      geoSvc    () const ;
  inline IGiGaKineCnvSvc*      kineSvc   () const ;
  inline IGiGaHitsCnvSvc*      hitsSvc   () const ;
  inline IDataProviderSvc*     evtSvc    () const ;
  inline IDataProviderSvc*     detSvc    () const ;
  inline IChronoStatSvc*       chronoSvc () const ;
  inline IGiGaSvc*             gigaSvc   () const ;
  inline IGiGaSetUpSvc*        setupSvc  () const ;
  inline IParticlePropertySvc* ppSvc     () const ;
  ...
};
```

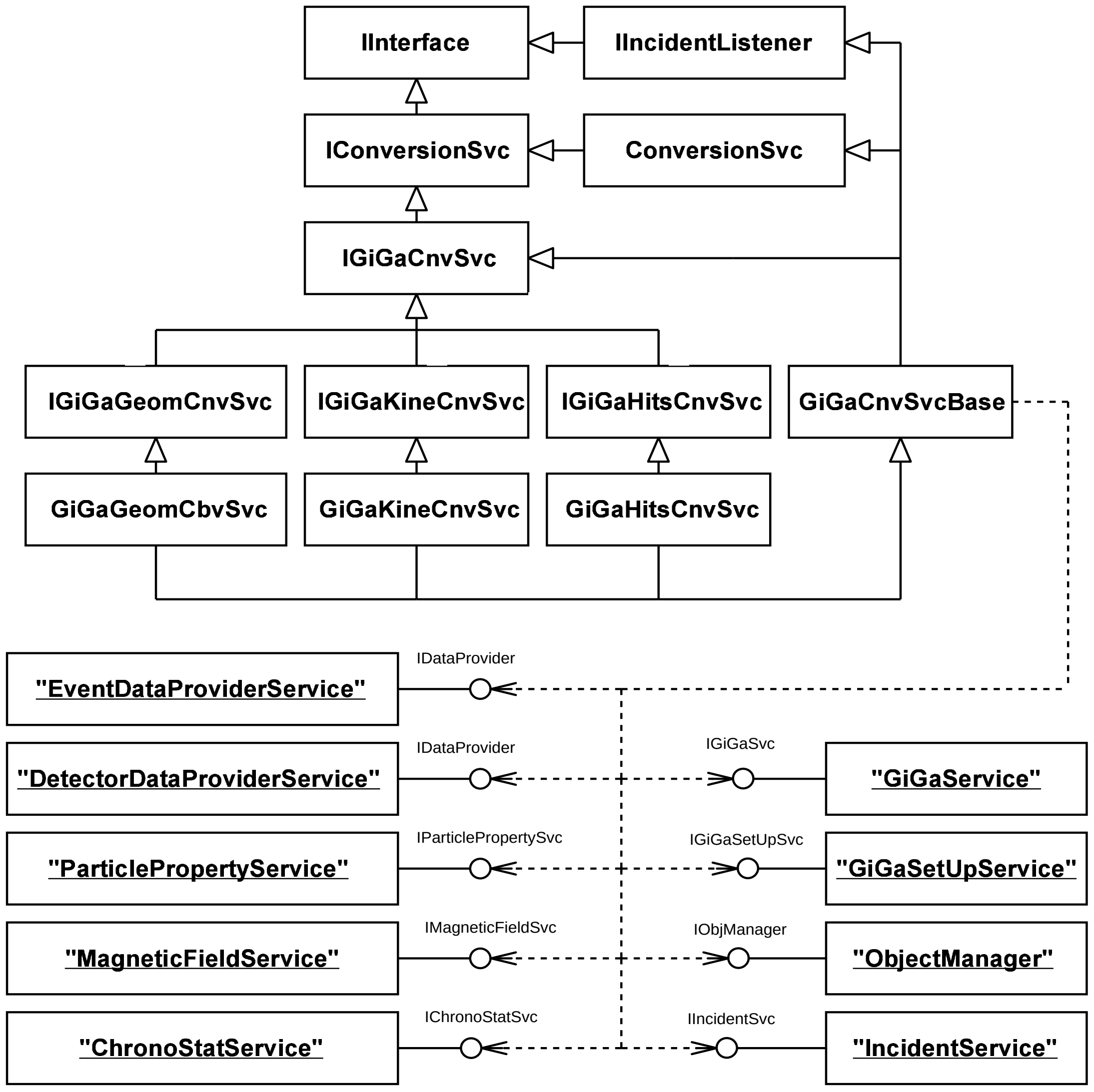

Figure 3.1: The simplified class diagram for GIGA **`Conversion Service`**s. Some base classes, like **`IProperty`**, **`IService`** and **`Service`** are not shown. All named components, located and used by base class **`GiGaCnvBase`** are explicitely indicated

## 3.2 Conversion of detector description

### 3.2.1 Geometry `Conversion Service`

Conversion of GAUDI detector description into GEANT4 objects is under the control of **`Conversion Service`** of type **`GiGaGeomCnvSvc`**. This **`Service`** implements the **`IGiGaGeomCnvSvc`** interfaces.

In addition to properties, inherited form **`GiGaCnvSvcBase`** class, it has several specific properties, listed in table 3.2

These properties includes the description of top volume where all GEANT4 geometry tree resides. This volume has box shape. Other its properties of this top-volume are under the control of properties of **`GiGaGeomCnvSvc`**.

Table 3.2: Specific properties of **`GiGaGeomCnvSvc`** and their default values.

| Property Name | Default Value |
|---|---|
| **`WorldMaterial`** | **`"/dd/Materials/Air"`** |
| **`WorldPhysicalVolumeName`** | **`"Universe"`** |
| **`WorldLogicalVolumeName`** | **`"World"`** |
| **`WorldMagneticField`** | **`""`** |
| **`XsizeOfWorldVolume`** | **`50*m`** |
| **`YsizeOfWorldVolume`** | **`50*m`** |
| **`ZsizeOfWorldVolume`** | **`50*m`** |

### 3.2.2 Conversion of materials

Materials are described in GAUDI DETDESC package with the help of 3 specific material classes: **`Element`**, **`Isotope`** and **`Mixture`**. These classes implements common **`Material`** interface. Since material description within DETDESC and GEANT4 are quite similar the conversion procedure becomes trivial, one-to-one transformation. The tabulated properties of materials are also converted into GEANT4.

Three concrete **`Converter`**s are implemented: **`GiGaIsotopeCnv`**, **`GiGaElementCnv`** and **`GiGaMixtureCnv`**. They inherits from helper **`GiGaCnvBase`** class.

#### Naming convention

Materials are converted into GEANT4 classes **`G4Material`**, **`G4Element`**, **`G4Mixture`** and **`G4Isotope`**. As a name a **`fullpath()`** of corresponding **`DataObject`** is used, e.g. **`"/dd/Materials/Air"`**.

### 3.2.3 Conversion of geometry objects

Geometry description in DETDESC package is made through 3 types of identifiable objects **LVolume**, **DetectorElement** and **Surface**[1]. The simplified class diagrams for 3 corresponding **Converter** classes **GiGaLVolumeCnv**, **GiGaDetectorElementCnv** and **GiGaSurfaceCnv** are shown on figure 3.2. Call-backs from geometry **Converter**s to **IGiGaGeomCnvSvc** interface are explicitly indicated.

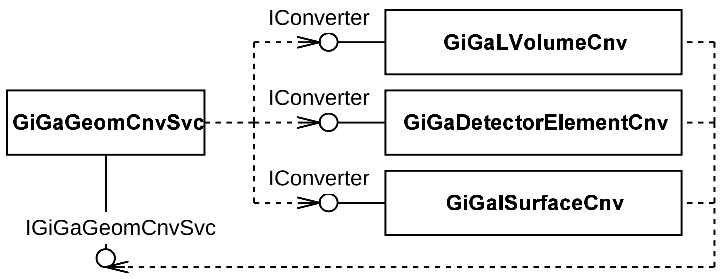

Figure 3.2: The simplified class diagrams for GIGA geometry **Converter**s. The common base class **GiGaCnvBase** is not shown on the figure.

These classes are converted into GEANT4 classes **G4LogicalVolume**, **G4PVPlacement**, **G4LogicalSkinSurface** and **G4LogicalBorderSurface**.

#### Naming convention

Logical volume (of type **G4LogicalVolume**) in GEANT4 get its name from **name()** method from **ILVolume** interface, which is the full address of logical volume in transient store, e.g. **"/dd/Geometry/LHCb/lvLHCb"**.

Situation with naming of physical volumes (of type **G4PVPlacement**) is a little bit more complicated. Physical volume gets the name of the form **"<MotherLVName>#PVname"** if it is created during conversion of its mother logical volume or **"FullPathForDetectorElement"** if it corresponds to detector element, which is converted in a separate way without conversion of higher level detector elements.

Surfaces (of types **G4LogicalSkinSurface** and **G4LogicalBorderSurface**) get their name from **fullpath()** method of **Surface** class, e.g. **"/dd/Geometry/Rich1/MirrorSurface"**. The corresponding **G4OpticalSurface** class gets the same name.

[1]used to describe the optical boundaries

## 3.3 Conversion of kinematics

Conversion of **LHCbEvent** data model into **Geant4** objects and from **Geant4**objects is under the control of **`Conversion Service`** of type **`GiGaKineCnvSvc`**. This **`Service`** implements the **`IGiGaKineCnvSvc`** interface. Two concrete converters are implemented:

- **`GiGaMCVertexCnv`** for conversion of container of **`MCVertex*`** objects into **Geant4** primary event and for conversion of **`G4TrajectoryContainer`** class into container of objects of type **`MCVertex*`**.
- **`GiGaMCParticleCnv`** for conversion of **`G4TrajectoryContainer`** class into container of objects of type **`MCParticle*`**.

### 3.3.1 Conversion of primary event

Primary event in **Geant4** is represented by **`G4PrimaryVertex`** and **`G4PrimaryParticle`** classes.

The container of **`MCVertex*`** objects is converted by **`GiGaMCVertexCnv`** **`Converter`** to these objects and the result of conversion is transferred to **`GiGaSvc`** using its streamer **`operator<<(G4PrimaryVertex*)`** method

One should pay the special attention to be sure that definitions of particles from **`IParticlePropetrySvc`** and particle definitions in **Geant4** are in agreement.

Unfortunately in current version of **Geant4** the information about proper decay time in primary event is lost. This feature is declared to be fixed soon.

### 3.3.2 Conversion of secondaries

The event record in **Geant4** is represented by **`G4TrajectoryContainer`** class, which is just an container of pointers to **`G4VTrajectory`** objects, which in turns contain information about trajectory points of pointers to **`G4VTrajectoryPoint`** objects.

**Geant4** tool kit provides users with two default ”standard” classes:

- **`G4Trajectory`** which implements **`G4VTrajectory`** interface
- **`G4TrajectoryPoint`** which implements **`G4VTrajectoryPoint`** interface

Unfortunately one could not convert these objects into **LHCbEvent** data model, since space points are not associated with time of flight information. Thats why two other classes are implemented:

- **`GiGaTrajectoryPoint`**
- **`GiGaTrajectory`**

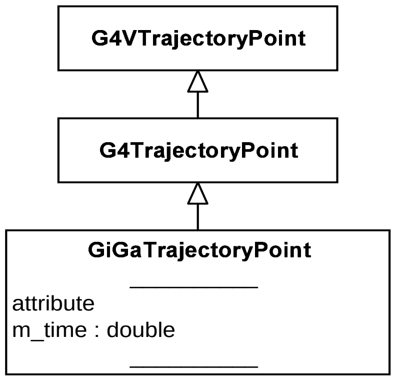

Figure 3.3: The class diagram for **`GiGaTrajectoryPoint`** class

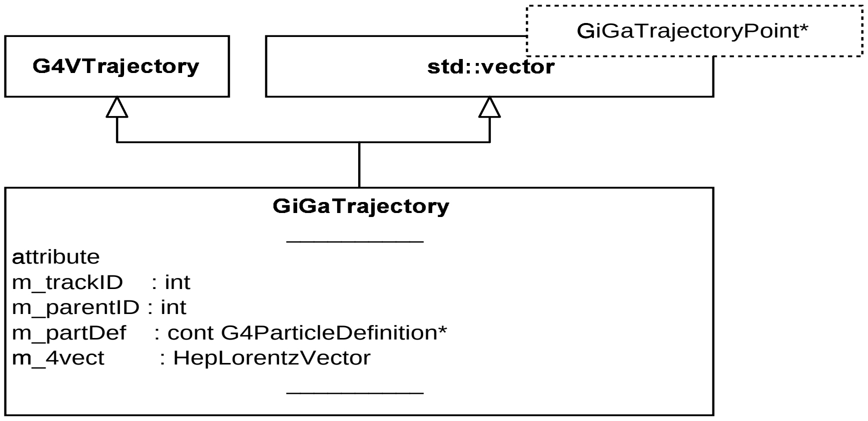

Figure 3.4: The class diagram for **`GiGaTrajectory`** class

The class diagrams for these classes are shown on figures 3.3 and 3.4.

**`G4TrajectoryContainer`** which contains the pointers to object of type **`GiGaTrajectory`** is converted into container of **`MCVertex*`** by **`GiGaMCVertexCnv Converter`** and to container of **`MCParticle*`** by **`GiGaMCParticleCnv Converter`**.

### Tracking Action

To be able to convert **GEANT4** event record into **LHCbEVENT** data model one need to use **`GiGaTrajectory`** class instead of default standard class **`G4Trajectory`**. It is achieved with the help of properly implemented **`GiGaTrackAction`** class.

## 3.4 Conversion of hits

Conversion of GEANT4 hits into LHCBEVENT data model is under the control of **`Conversion Service`** of type **`GiGaHitsCnvSvc`**. This **`Service`** implements the **`IGiGaHitsCnvSvc`**.

For each type of hit objects in GAUDItransient store a corresponding converter from GEANT4 hits is to be implemented. This part is sub-detector dependent.

### 3.4.1 Relations between ”tracks” and hits

Almost for all types of hits one need to preserve the relations between hits and ”tracks”. The essential point is, one could use the track ID (from **`G4Track`** object) inside hit object to keep such relation, but but one should take care that the corresponding **`G4Track`** object is marked to be saved into **`GiGaTrajectory`**. It is especially important for showering devices, where definitely not all **`G4Track`**s could be saved into corresponding objects of type **`GiGaTrajectory`**. The saving of the **`G4Track`** into **`GiGaTrajectory`** is under the control of corresponding class **`GiGaTrackAction`**. One could force the saving of particular tracks by using helper class **`GiGaTrackInformation`**, which implements **`G4VUserTrackInformation`** interface. Object of this type is attached to each **`G4Track`** object and could be used to mark track to be saved.

## 3.5 Streams

To simplify the definitions of input data flows of **`GiGaSvc`** an utility class **`GiGaStream`** is developed. It is not mandatory for usage with GIGA but it is helpful for providing **`GiGaSvc`** with input data. Essentially it is just an **`Algorithm`** which gets the data from transient store and puts them into **`GiGaSvc`** using defined **`Conversion Service`**s.

Typically one needs to define 2 such input streams - one for geometry objects, to be initialised in the begin of the job and one stream for kinematics, to be executed every event.

**`GiGaStream`** class inherits from **`Algorithm`** and in addition to standard properties it has specific properties which are listed in table 3.3.

Table 3.3: Specific properties of **`GiGaStream`** and their default values.

| Property Name | Default Value |
|---|---|
| **`"ExecuteOnce"`** | **`false`** |
| **`"ConversionSvcName"`** | **`"GiGaKineCnvSvc"`** |
| **`"DataProviderSvcName"`** | **`"EventDataSvc"`** |
| **`"DataManagerSvcName"`** | **`"EventDataSvc"`** |
| **`"StreamItems"`** | empty |
| **`"FillGiGaStream"`** | **`true`** |

And the example configuration of the job could be:

```
/// add streams
ApplicationMgr.TopAlg +=  { "GiGaStream/Kine" };
/// configure input kinematics stream
Kine.StreamItems         = { "/Event/MC/Generator/MCVertices" };
///
```

## 3.6 **GiGaSvc** as persistency

**GiGa** could be naturally considered as an artificial persistency of ”read-only” technology. This is illustrated by simplified interaction diagramm shown on figure 3.5. The illustrated example shows the simplified interaction between data provider (**"EventDataSvc"**), its persistensy service (**"EventPersistencySvs"**), conversion service(**"GiGaKineCnvSvc"**), concrete converter (**"GiGaMCParticleCnv"**) and **GiGaSvc**. In this context **GiGaSvc** is considered just external data stream, and concrete converter just ”reads” the appropriate data from that stream using ordinary **operator>>**.
The job should be properly configured to use this possibility, e.g.

```
/// ...
/// add conversion services to Event Persistency Service
EventPersistencySvc.CnvServices += { "GiGaKineCnvSvc" } ;
EventPersistencySvc.CnvServices += { "GiGaHitsCnvSvc" } ;
/// ...
```

This chain could be triggered by following lines inside user **Algorithm**:

```
///  ...
///  remember: typedef ObjectVector<MCParticle> MCParticleVector;
const std::string address("/Event/MC/G4/MCParticles");
SmartDataPtr<MCParticleVector> mcpv( eventSvc() , address );
/// check "mcpv" and then use it!
/// ...
```

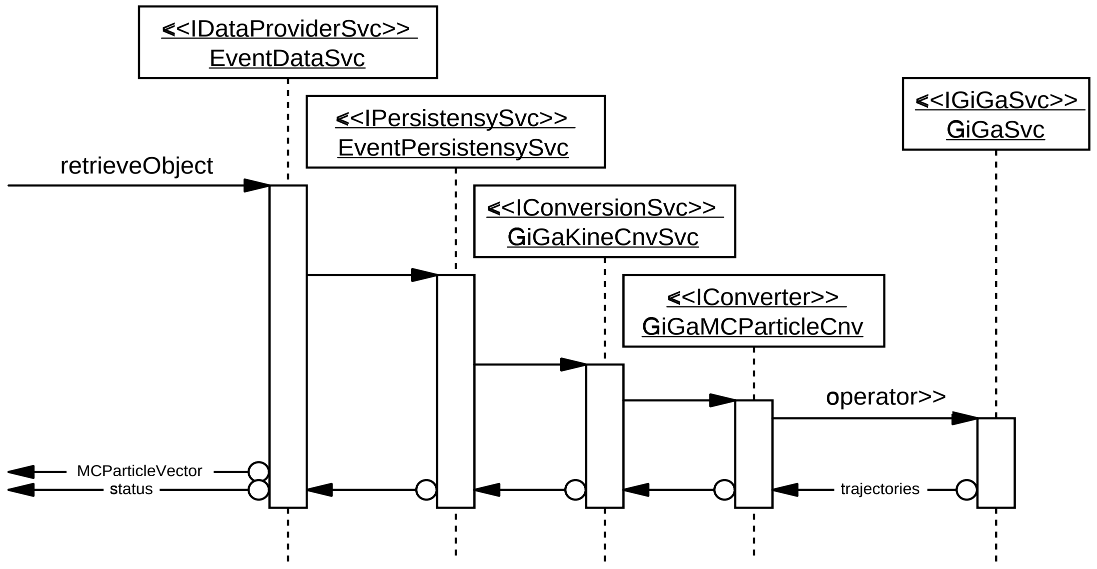

Figure 3.5: The simplified interaction diagram for persistency emulation with **GiGa**

# Chapter 4: Configuration classes

## 4.1 Physics List

GEANT4 requires to be provided with concrete implementation of abstract **`G4VUserPhysics-List`** interface.

GIGA provides the posibility of instantiation of such objects using abstract factories approach. To get this behaviour the intermediate abstract interface **`IGiGaPhysList`** is defined by merging **`G4VUserPhysicsList`** and **`IInterface`** interfaces. Then one could reuse the standard approach of abstract factories to instantiate such objects. The simplified class diagrams are shown on figure 4.1.

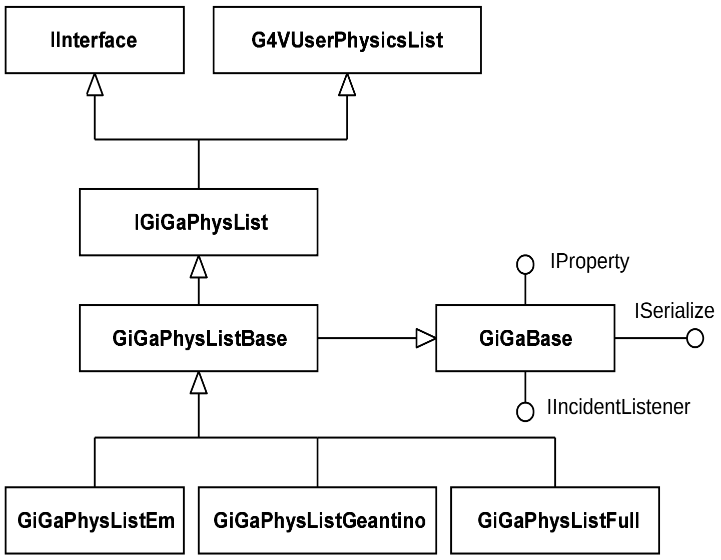

Figure 4.1: The class diagrams for **`GiGaPhysList`** classes

Three concrete physics list classes are implemented:

- **`GiGaPhysListGeantino`**
  Defines geantino as particle and transportation processes
- **`GiGaPhysListEm`**
  Defines electromagnetic particles and typical set of electromagnetic processes
- **`GiGaPhysListFull`**
  Defines quite large set of particles and processes, corresponds to standard example **`N04`** from GEANT4 distribution.

Other concrete physics lists could be implemented(e.g. to deal with optical photons) by inheritance from **`GiGaPhysListBase`** class.
The instantiation of physics list classes are under the control of **`GiGaSvc`** and in managed through properties of **`GiGaSvc`**:

```
/// ...
/// physics list to be instantiated
GiGaSvc.PhysicsList = "GiGaPhysListEm/EmPhysList";
/// configure physics list, set e.g. default cut value in mm
EmPhysList.Cut = 50.0;
/// ...
```

**GEANT4** takes care about correct deletion of such object.

Additional with respect to the base class **`GiGaBase`** properties of **`GiGaPhysListBase`** are listed in table 4.1.

Table 4.1: Specific properties of **`GiGaPhysListBase`** and their default values.

| Property Name | Default Value |
|---|---|
| **`"Cut"`** | **`2.0*mm`** |

## 4.2 Magnetic Field

Magnetic field for GEANT4 must be provided with concrete implementation of abstract **`G4MagneticField`** interface.

GIGA provides the possibility of instantiation of such objects using abstract factories approach. To get this behaviour the intermediate abstract interface **`IGiGaMagField`** is defined by merging **`G4MagneticField`** and **`IInterface`** interfaces. Then one could reuse the standard approach of abstract factories to instantiate such objects. The simplified class diagrams are shown on figure 4.2.

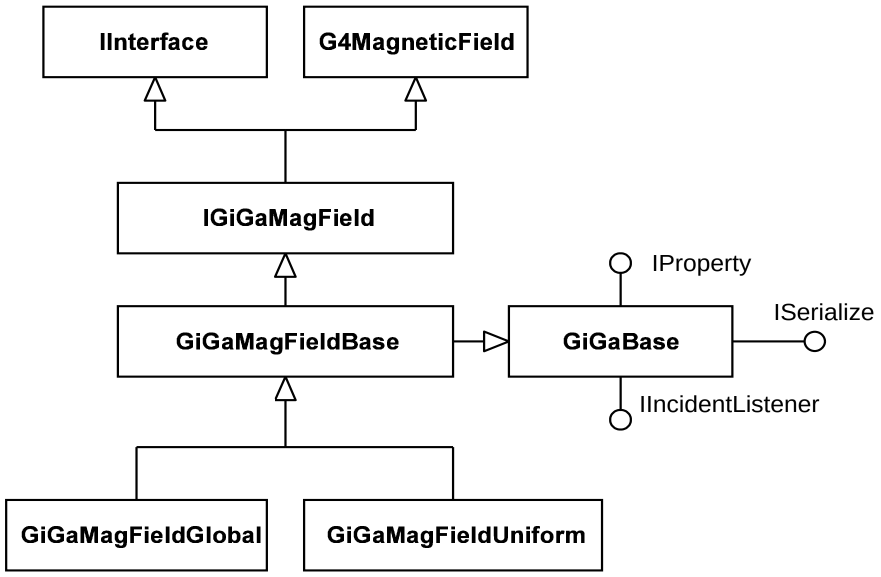

Figure 4.2: The class diagrams for **`GiGaMagField`** classes

Two concrete magnetic field classes are implemented: **`GiGaMagFieldGlobal`** and **`GiGaMagFieldUniform`**. the first one is just a delegation to **`IMagneticFieldSvc`**. The second one represents the uniform field with components, which could be set by propertied of corresponding class (see table 4.2).

Table 4.2: Specific properties of **`GiGaMagFieldUniform`** and their default values.

| Property Name | Default Value |
|---|---|
| **`"Bx"`** | **`0.0`** |
| **`"By"`** | **`0.0`** |
| **`"Bz"`** | **`0.0`** |

One could easily implement any other magnetic fields using inheritance from **`GiGaMagFieldBase`** class.

There exist 2 ways to use magnetic field in GEANT4:

- define the global field fo the whole set up
- redefine the magnetic field for some logical volume (e.g. magnet yoke)

The specific magnetic field for logical volume should be declared in the description of logical volume and it would be instantiated and associated with logical volume during the conversion procedure of logical volume to **`G4LogicalVolume`** class:

```
<!-- XML description of logical volume -->
 <logvol  name = .........
    magfield = 'GiGaMagFieldUniform/MagFieldInYoke'
 </logvol>
```

The global magnetic field is the property of **`GiGaGeomCnvSvc`** and could be configured through e.g job options technique:

```
/// ...
/// declare constant magnetic field as global field
GiGaGeomCnvSvc.WorldMagneticField = "GiGaMagFieldUniform/Uniform";
/// confiugure magnetic field
Uniform.Bx =  0.0;
Uniform.By = 10.0;
Uniform.Bz = 10.0;
/// ...
```

## 4.3 Description of sensitive detector

Sensitive detector description in GEANT4 must be done with concrete implementations of abstract **`G4VSensitiveDetector`** interface.

GIGA provides the possibility of instantiation of such objects using abstract factories approach. To get this behaviour the intermediate abstract interface **`IGiGaSensDet`** is defined by merging **`G4VSensitiveDetector`** and **`IInterface`** interfaces. Then one could reuse the standard approach of abstract factories to instantiate such objects. The simplified class diagrams are shown on figure 4.3.

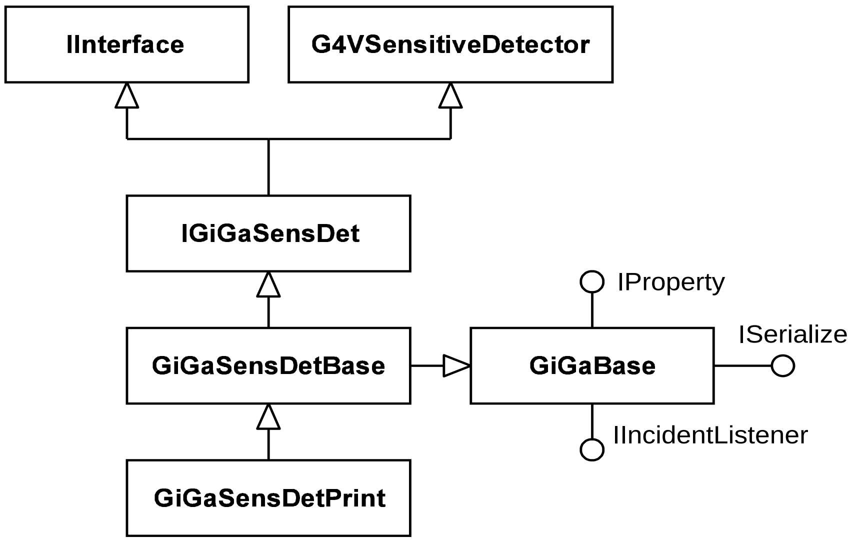

Figure 4.3: The class diagram for **`GiGaSensDetBase`** class

Each concrete sensitive detector must be implemented by inheritance form **`GiGaSensDet-Base`** class. Example of primitive implementation of sensitive detector is **`GiGaSensDet-Print`** class, which just performs the printout of step information when particle crosses the sensitive detector without creation of hits.

The specific properties of **`GiGaSensDetBase`** class are listed in table 4.3.

Table 4.3: Specific properties of **`GiGaSensDetBase`** and their default values.

| Property Name | Default Value |
|---|---|
| **`"DetectorElement"`** | **`"/dd/Structure/LHCb"`** |
| **`"Active"`** | **`true`** |

Concrete sensitive detector is a property of logical volume and it is instantiated automatically during conversion procedure of logical volume into **`G4LogicalVolume`** class:

```
<!-- XML description of logical volume -->
 <logvol  name = .........
    sensdet = 'GiGaCaloDetector/EcalDet'
 </logvol>
```

It is worth to mention that

- There could be several instances of the same sensitive detector, e.g. 4 instances of calorimeter sensitive detector, one per each calorimeter subsystem ( **`Prs`**,**`Spd`**,**`Ecal`** and **`Hcal`**)
- Several logical volumes could be associated with the same instance of sensitive detector, e.g. Scintillator tiles and absorber plates in electromagnetic calorimeter would be associated with the same instance of sensitive detector
- each sensitive detector has association with certain detector element (default - the whole **`"/dd/Structure/LHCb"`** detector). Such association is used to register sensitive detector in sensitive detector manager with appropriate path

It is worth to set a convention of the name of hits collections which are created by a specific sensitive detector. Such convention would be very useful for automatic creation of **`IOpaqueAddress`** during the conversion of hits.

... to be continued ...

## 4.4 Event action

Specific event action description in GEANT4 could be done with concrete implementations of **`G4UserEventAction`** class.

GIGA provides the posibility of instantiation of such objects using abstract factories approach. To get this behaviour the intermediate abstract interface **`IGiGaEventAction`** is defined by merging **`G4UserEventAction`** class with **`IInterface`** interface. Then one could reuse the standard approach of abstract factories to instantiate such objects. The simplified class diagrams are shown on figure 4.4.

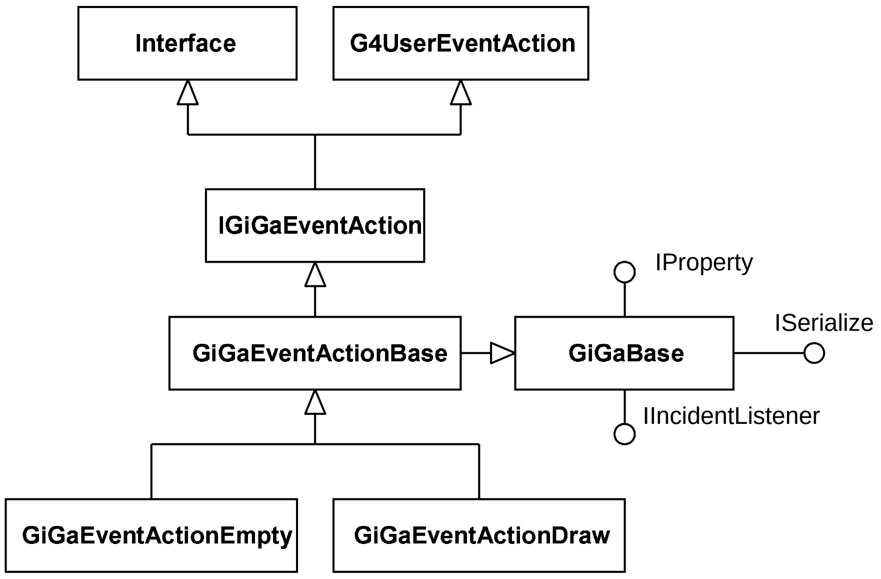

Figure 4.4: The class diagram for **`GiGaEventAction`** classes

Each concrete event action must be implemented by inheritance form **`GiGaEventAction-Base`** class. Example of primitive (just empty) implementation of event action is **`GiGaEven-tActionEmpty`** class, which does nothing. A little bit less trivial imeplemenation is **`Gi-GaEventActionDraw`** class which at the end of event processing triggers the visualization of **`G4Event`**, which is just the visualisation of all saved trajectories, hits an digits. It is worth to mention that only stored trajectories are visualized.

The event action is the property of **`GiGaSvc`** and could be configured through e.g job options technique :

```
/// ...
/// declare the stepping action:
GiGaSvc.EventAction = "GiGaEventActionDraw/DrawEvent";
/// ...
```

## 4.5 Tracking action

Specific tracking action description in GEANT4 ccould be done with concrete implementations of **`G4UserTrackingAction`** class.

GIGA pprovides the possibility of instantiation of such objects using abstract factories approach. To get this behaviour the intermediate abstract interface **`IGiGaTrackAction`** is defined by merging **`G4UserTrackingAction`** class with **`IInterface`** interface. Then one could reuse the standard approach of abstract factories to instantiate such objects. The simplified class diagrams are shown on figure 4.5.

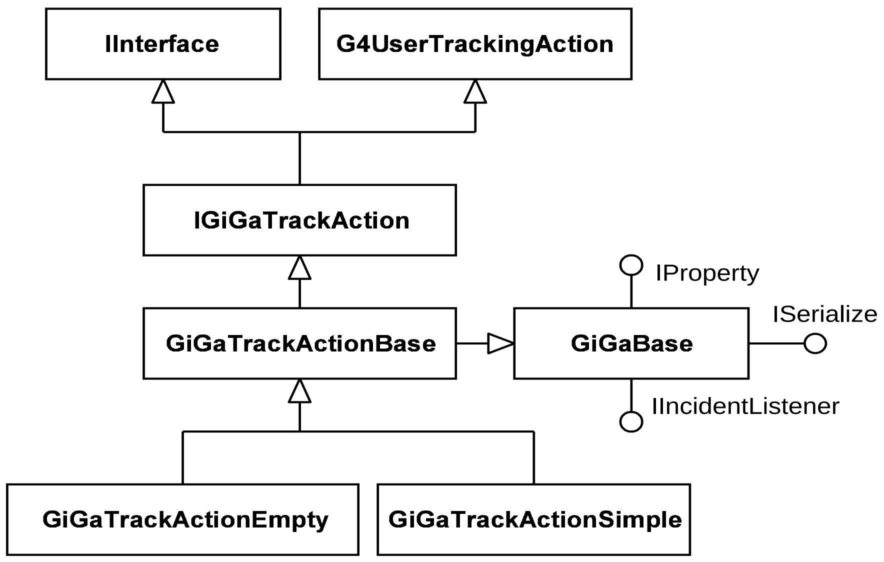

Figure 4.5: The class diagrams for **`GiGaTrackAction`** classes

Two concrete tracking action classes are implemented

- **`GiGaTrackActionEmpty`**
  It is just primitive empty tracking action
- **`GiGaTrackActionSimple`**
  It is quite simple tracking action which creates **`GiGaTrajectories`** and provides possibility to suppress the saving into trajectories **`G4Track`**s, which are ”out of interest”

Each concrete tracking action must be implemented by inheritance form **`GiGaTrackActionBase`** class.
The tracking action is the property of **`GiGaSvc`** and could be configured through e.g job options technique:

```
/// ...
/// declare the tracking action:
GiGaSvc.TrackingAction = "GiGaTrackActionSimple/SimpleTrack";
/// configure tracking action
```

It is worth to mention that if one needs to convert the secondaries from **GEANT4** to **LHCBEVENT** data model objects, one **must** exploit the non trivial tracking action like **`GiGaTrackActionSimple`** to save tracks into objects of type **GiGaTrajectory**.

### 4.5.1 `GiGaTrackActionSimple` class

**`GiGaTrackActionSimple`** class is an a simple example of non trivial tracking action. It allows:

- The saving of **`G4Track`** objects into **`GiGaTrajectory`** objects, thus allowing the conversion from **GEANT4**to **LHCBEVENT** data model to be performed.
- Take a full control over saving of tracks.

Specific properties of class **`GiGaTrackActionSimple`** are listed in table 4.4. The instance of this class could be configured through e.g job options technique :

```
/// ...
/// declare the tracking action:
GiGaSvc.TrackingAction = "GiGaTrackActionSimple/SimpleTrack";
/// configure tracking action
/// store ANY muons
SimpleTrack.StoreByOwnType     = true ;
SimpleTrack.StoredOwnTypes     = { "mu+" , "mu-" };
/// store all particles with kinetic energies > 1 GeV
SimpleTrack.StoreByOwnEnergy   = true ;
SimpleTrack.OwnEnergythreshold = 1.0*GeV ;
/// store all particles which produce muons:
SimpleTrack.StoreByChildType     = true ;
SimpleTrack.StoredChildTypes     = { "mu+" , "mu-" };
/// ...
```

Table 4.4: Specific properties of **`GiGaTrackActionSimple`** and their default values.

| Property Name | Default Value |
|---|---|
| **`"StoreAll"`** | **`false`** |
| **`"StorePrimary"`** | **`true`** |
| **`"StoreByOwnEnergy"`** | **`false`** |
| **`"StoreByOwnType"`** | **`false`** |
| **`"StoreByChildEnergy"`** | **`false`** |
| **`"StoreByChildType"`** | **`false`** |
| **`"StoreMarkedTracks"`** | **`true`** |
| **`"OwnEnergyThreshold"`** | **`10*TeV`** |
| **`"ChildEnergyThreshold"`** | **`10*TeV`** |
| **`"StoredOwnTypes"`** | empty |
| **`"StoredChildTypes"`** | empty |

**`G4Track`** object is saved into **`GiGaTrajectory`** object if at least one condition is fulfilled:

- **`"StoreAll"`** property of **`GiGaTrackActionSimple`** instance is activated

- **`"StoreAll"`** property is activated and track is an primary particle
- **`"StoreByOwnEnergy"`** property is activated and the track has kinetic energy over threshold, which is set through **`"OwnEnergyThreshold"`** property
- **`"StoreByOwnType"`** property is activated and the track has a type corresponding to one of the types, which are set through **`"StoredOwnTypes"`** property
- **`"StoreByChildEnergy"`** property is activated and the track has at least one child track with kinetic energy over threshold, which is set through **`"ChildEnergyThreshold"`** property
- **`"StoreByOwnType"`** property is activated and the track has at least one child of the type corresponding to one of the types, which are set through **`"StoredChildTypes"`** property
- **`"StoreMarkedTracks"`** property is activated and the track is marked as **`"toBeStored"`** using **`GiGaTrackInformation`** class

If **`G4Track`** is decided not to be stored, the **`ParentID`** field for all its daughter particles are set to be it's own saved parent track, thus keeping the whole event history information to be consistent.

... to be continued ...

## 4.6 Stacking action

Specific stacking action description in GEANT4 could be done with concrete implementations of **`G4UserStackingAction`** class.

GIGA provides the posibility of instantiation of such objects using abstract factories approach. To get this behaviour the intermediate abstract interface **`IGiGaStackAction`** is defined by merging **`G4UserStackingAction`** class with **`IInterface`** interface. Then one could reuse the standard approach of abstract factories to instantiate such objects. The simplified class diagramms are shown on figure 4.6.

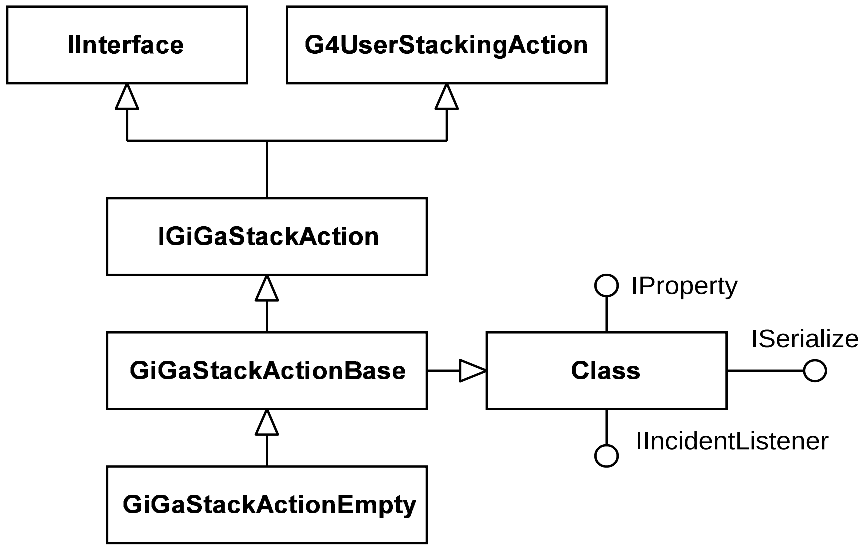

Figure 4.6: The class diagram for **`GiGaStackActionBase`** class

Each concrete stacking action must be implemented by inheritance form **`GiGaStackActionBase`** class. Example of primitive (just empty) implementation of stacking action is **`GiGaStackActionEmpty`** class, which does nothing.
The stepping action is the property of **`GiGaSvc`** and could be configured through e.g job options technique:

```
/// ...
/// declare the stepping action:
GiGaSvc.StackingAction = "GiGaStackActionEmpty/EmptyStack";
/// ...
```

## 4.7 Stepping action

Specific stepping action description in GEANT4 could be done with concrete implementations of **`G4UserSteppingAction`** class.

GIGA provides the posibility of instantiation of such objects using abstract factories approach. To get this behaviour the intermediate abstract interface **`IGiGaStepAction`** is defined by merging **`G4UserStepingAction`** class with **`IInterface`** interface. Then one could reuse the standard approach of abstract factories to instantiate such objects. The simplified class diagrams are shown on figure 4.7.

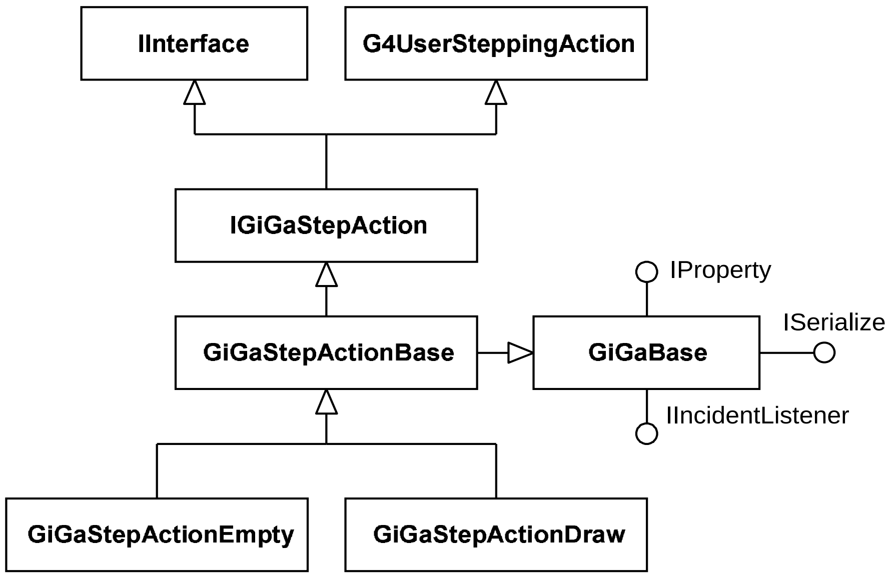

Figure 4.7: The class diagram for **`GiGaStepAction`** classes

Each concrete stepping action must be implemented by inheritance form **`GiGaStepActionBase`** class. Example of primitive (just empty) implementation of stepping action is **`GiGaStepActionEmpty`** class, which does nothing. Less trivial example is **`GiGaStepActionDraw`** class, which performs visualisation of each step. It is worth to mention that it differs from visualization of trajectories.
The stepping action is the property of **`GiGaSvc`** and could be configured through e.g job options technique :

```
/// ...
/// declare the stepping action:
GiGaSvc.SteppingAction = "GiGaStepActionDraw/DrawStep";
/// ...
```

## 4.8 Helper `GiGaBase` class

All configuration classes in GIGA uses the common helper base **`GiGaBase`**. The simplified class diagram for this helper class is shown on figure 4.8.

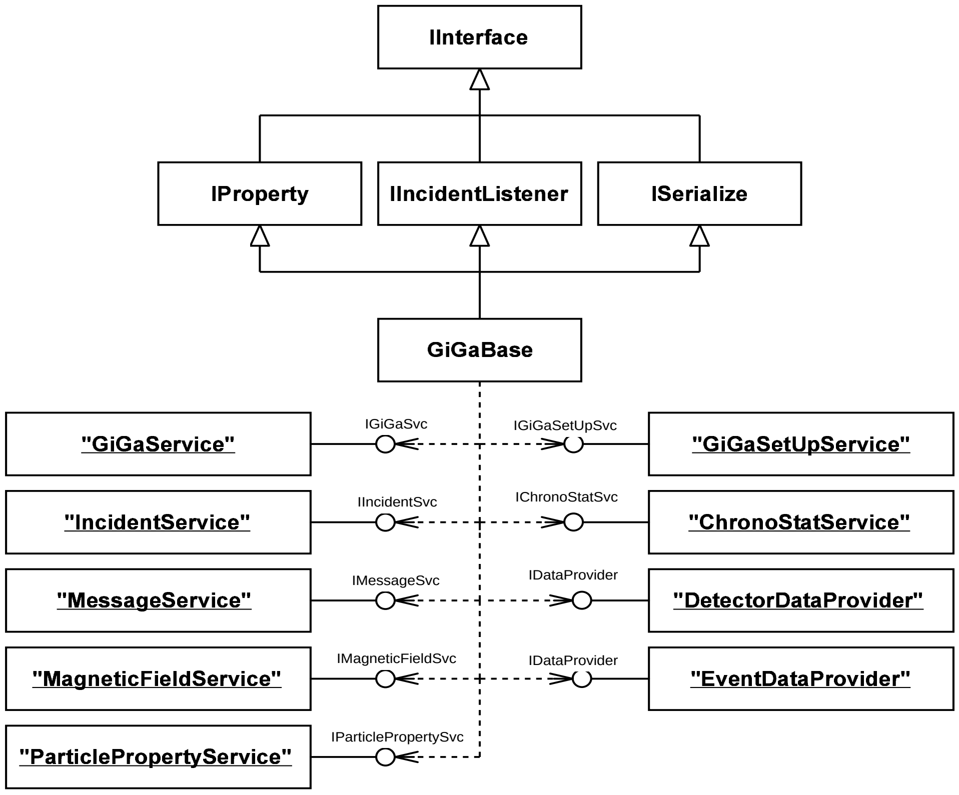

Figure 4.8: The simplified class diagram for helper **`GiGaBase`** class. The named components, located and used by **`GiGaBase`** are explicitely shown

The standard properties of **`GiGaBase`** class and their default values are listed in table 4.5.
The base class **`GiGaBase`** locates basic interfaces of the named components using property information and provides with following accessors:

```
class GiGaBase: .....
  ...
protected:
  ///  Accesors to needed services and Service Locator
  inline ISvcLocator*           svcLoc    () const ;
  inline IGiGaSvc*              gigaSvc   () const ;
  inline IGiGaSetUpSvc*         setupSvc  () const ;
  inline IMessageSvc*           msgSvc    () const ;
  inline IChronoStatSvc*        chronoSvc () const ;
  inline IDataProviderSvc*      evtSvc    () const ;
  inline IDataProviderSvc*      detSvc    () const ;
  inline IIncidentSvc*          incSvc    () const ;
  inline IParticlePropertySvc*  ppSvc     () const ;
  inline IMagneticFieldSvc*     mfSvc     () const ;
  ....
};
```

Table 4.5: Properties of **GiGaBase** and their default values.

| Property Name | Default Value |
|---|---|
| **"OutputLevel"** | **MSG::NIL** |
| **"GiGaService"** | **"GiGaSvc"** |
| **"GiGaSetUpService"** | **"GiGaSvc"** |
| **"MessageService"** | **"MessageSvc"** |
| **"ChronoStatService"** | **"ChronoStatSvc"** |
| **"EventDataProvider"** | **"EventDataSvc"** |
| **"DetectorDataProvider"** | **"DetectorDataSvc"** |
| **"IncidentService"** | **"IncidentSvc"** |
| **"ParticlePropertyService"** | **"ParticlePropertySvc"** |
| **"MagneticFieldService"** | **"MagneticFieldSvc"** |

If property value is set to be empty name (**""**) the corresponding component is not required to be located.

## List of Figures

## List of Tables