European Laboratory for Particle Physics
Laboratoire Européen pour la Physique des Particules
CH-1211 Genève 23 - Suisse

The Gaudi Reflection Tool

Crash Course

Document Version:	1
Document Date:	4. Sept. 2001
Document Status:	Draft
Document Author:	Stefan Roiser

Abstract

1 Introduction

The Gaudi Reflection Tool is a model for the metarepresentation of C++ classes. This model has to be filled with the corresponding information to each class that shall be represented through this model. Ideally the writing of this information and filling of the model will be done during the creation of the real classes. Once the model is filled with information a pointer to the real class will be enough to enter this model and retrieve information about the real class.

If there are any relations to other classes, and the information about these corresponding classes is also part of the metamodel one can jump to these classes and also retrieve information about them. So an introspection of C++ classes from an external point of view can be realised quite easily.

2 The Meta Model

The struture of the Gaudi Reflection Model is divided into four parts. These four parts are represented by four C++-classes. The corepart is the class called *MetaClass*. Within this class everything relevant to the real object will be stored. This information includes the name, a description, the author, the list of methods, members and constructors of this class etc. This class is also the entrypoint to the MetaModel. Using the MetaClass function forName and the string of the type of the real class as an argument, a pointer to the corresponding instance of MetaClass will be returned. With this pointer every possible information about the class can be retrieved.

This includes information about the fields of the class. Information about these fields is stored in the class MetaField. With a pointer to an instance of MetaField information about this field can be retrieved. This information includes the name of the field, a description, the offset relative to the beginning of the class and its modifiers (eg static, public, etc.). With a pointer to the MetaField also the value of this field can be retrieved and also set if one knows its type. (The type is again a class and can be retrieved by the MetaClass function ‘type’).

The other two classes describing the construtors and methods of the MetaModel are not implemented yet and so will not be discussed any further.

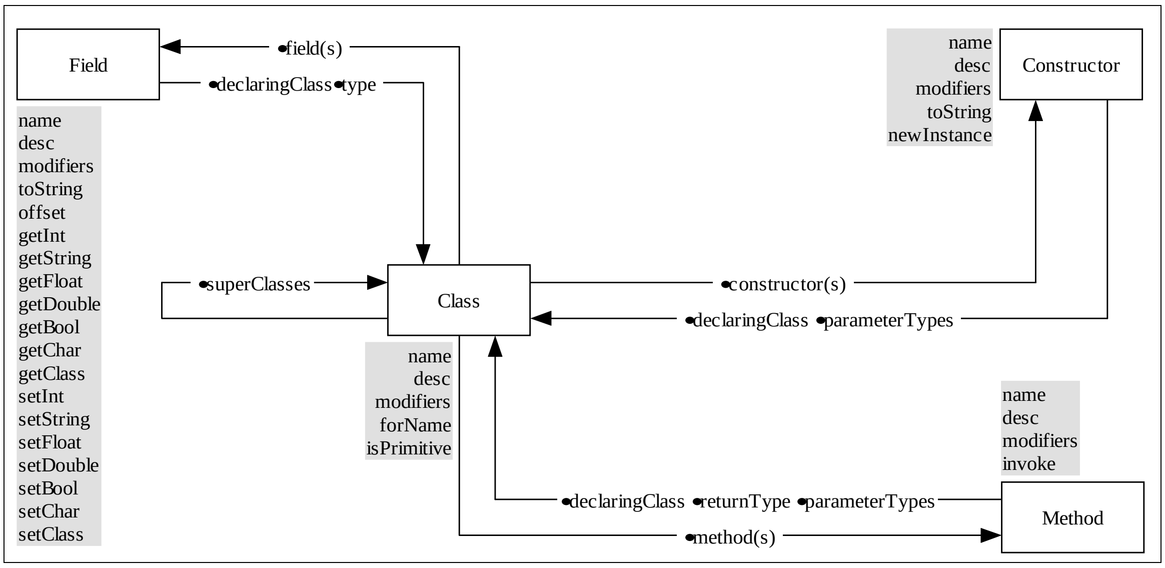

Figure 1 The structure of the Metamodel

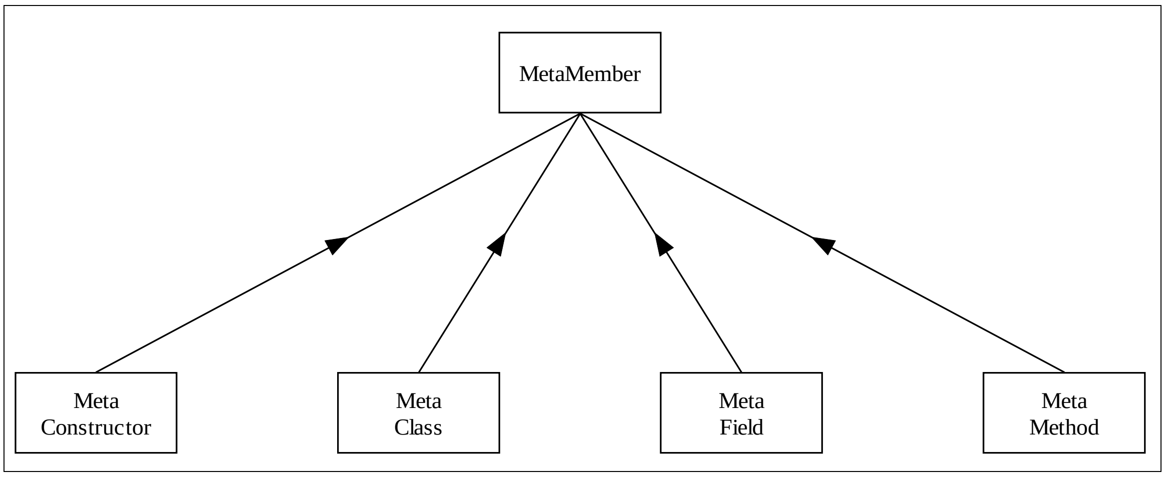

Figure 2 The C++ structure of the Metamodel

3 Tutorial for retrieval and manipulation of information

3.1 Examining Classes

3.1.1 Retrieving MetaClass Objects

The only way how one can retrieve the pointer to a MetaClass instance is by the static MetaClass function ‘forName’. The input paramter of this function is a string representing the name of the real class. But how to get the name of this real class if one has only a pointer to an object and no knowledge to which class this pointer belongs to?

Fortunately there is a function called typeid in C++ which takes a pointer as an argument and returns a struct called type_info. This struct bears also the name of the function the pointer belongs to.

This will also work if the pointer was casted to a baseclass of this instance. So in case of the Gaudi transient event store one can easily convert a pointer to type ‘DataObject’ and the information about the original class will still remain in the struct type_info.

Listing 1 Retrieving a pointer to MetaClass

```
  1:  const type_info& ti = typeid(*entrypoint);
  2:  MetaClass* mc = MetaClass::forName(System::typeinfoName(ti));
```

Listing 1 gives an example on how to retrieve a pointer to MetaClass. Line 1 creates a reference to type_info assigning it the typeid of ‘entrypoint’ which is a pointer to the object. In Line 2 the name of the struct type_info will be taken as an argument for the static MetaClass function ‘forName’ which returns the pointer to the corresponding MetaClass.

3.1.2 Getting information about the class

Once a pointer to the MetaClass is retrieved every other information can be shown. This includes the name of the class, the description of the class and its author. Assuming we already have retrieved a pointer to a MetaClass called mc, Listing 2 gives a short example of how to retrieve this information. It’s result is are shown in

Listing 2 Retrieving information about the class

```
std::cout << “ClassInfo:“ << std::endl
          << “Name:\t“ << mc->name() << std::endl
          << “Author:\t“ << mc->author() << std::endl
          << “Description:\t“ << mc->desc() << std::endl
```

Listing 3 Results of Listing 2

```
ClassInfo:
Name:        MCEvent
Author:      Pavel Binko
Description: Stores essential information of the Monte Carlo event
```

3.1.3 Retrieving BaseClasses

Retrieving the baseclasses of a given class is similar to retrieving other information about an object. With the MetaClass method ‘superClasses’ a vector of pointers to MetaClasses will be returned. By walking this vector one can retrieve one pointer after the other and with the pointer to the MetaClass the game will start again.

Listing 4 Retrieving BaseClasses

```
std::vector<MetaClass*> mcv = mc->superClasses();
for (int i = 0; i < mcv.size(); ++i)\
{
    std::cout << “Baseclass of class “ << mc->name() << “: “
              << mcv[i]->name() << std::endl;
}
```

Listing 5 Results of Listing 4

```
Baseclass of class MCEvent: DataObject
```

3.1.4 Retrieving Class Fields

Similarly to the way of retrieving the information of Baseclasses, information about fields of the object in question can be retrieved. The MetaClass method ‘fields’ returns a vector of pointers to MetaFields.

Going through this vector returns one pointer after the other to the metainformation about the fields of the object.

Listing 6

```
std::vector<MetaField*> mfs = mc1->fields();
for (int i = 0; i < mfs.size(); ++i)
{
    std::cout << “Field” << i << “: “ << mfs[i]->name() << std::endl;
}
```

Listing 7 Results of Listing 6

```
Field1: m_pileUp
Field2: m_subMCEvents
Field3: m_lumi
```

3.2 Manipulating Objects

3.2.1 Getting Field information

Once a pointer to a MetaField is retrieved, information about the corresponding field may be gained through this pointer. Like the information about the class most of this general information are simple strings. Listing 8 shows some of the functions that can be used. (The whole list of functions can be seen in Appendix A on page page 9).

Listing 8

```
MetaField* mf1 = new MetaField();
mf1 = mc->field(“m_lumi”);
std::cout << “FieldInfo:“ << std::endl << “Name:\t” << mf1->name()
          << “Type:\t” << mf1->type()->name() << “Description:\t”
          << mf1->desc() << std::endl;
```

Listing 9

```
FieldInfo:
Name:        m_lumi
Type:        double
Description: Pile up flags
```

If one knows that the field ‘m_lumi’ is of type ‘double’ it is also possible to retrieve its value. If the type of the field is not known in advance it is also possible to get the type of the field as a string (as shown in Listing 8) and retrieve the value in a second step.

Listing 10 Retrieving a value of a field

```
std::cout << “This is the value of field “ << mf1->name() << “: “
          << mf1->getDouble(entrypoint) << std::endl;
```

Listing 11 Result of Listing 10

```
This is the value of field m_lumi: 3432
```

3.2.2 Setting values of fields

With the Gaudi Reflection Tool it is not only possible to get values of fields but also to set them. Listing 12 shows how one can do that.

Listing 12 Setting field values

```
mf1->setDouble(entrypoint, 2343);
std::cout << “The new value of field “ << mf1->name() << “ is: “
          << mf1->getDouble(entrypoint);
```

Listing 13 Results of Listing 12

```
The new value of field m_lumi is: 2343
```

4 How the MetaModel is organized

4.1 Filling the model

For the filling of the MetaModel some C++-functions are needed which set the relevant information about the class, its fields, methods and constructors. A complete (Pseudo)-C++-code for filling the MetaModel with the information of one class can be seen in Appendix B (on page 13).

4.2 Methods setting information of a class

The setting of information about a class will be passed to the constructor of the MetaClass-class. Inside this constructor several functions will be called which set the appropriate information. The call of the constructor can be seen in Listing 14. The arguments of this call are:

1. The name of the class
2. The description of the class

Additionally to the call of the constructor the name of the class with the pointer to its metaclass will be added to a static map, which is responsible for the retrieval of the pointer to the MetaClass.

Listing 14 Setting MetaClass information

```
MetaClass* mc = new MetaClass(“MCEvent”, “Stores essential
information of the Monte Carlo Event”);
MetaClass::addForName(“MCEvent”, mc);
```

Inside the Constructor (see Listing 15) of the MetaClass the two functions for setting the name of the class and its description are called. The third function ‘resolvePending’ is called periodically to solve problems when a previous found type of a field or baseclass could not be resolved. This function then tries to resolve these pending types.

Listing 15 Constructor of MetaClass

```
MetaClass(std::string name, std::string desc)
{
    setName(name);
    setDesc(desc);
    MetaClass::resolvePending();
}
```

4.3 Methods setting information of a Field

When setting the information for a field everything can be done by calling the constructor of the field. An example for this call is shown in Listing 16. This call of the constructor has 5 arguments which are:

1. The name of the field
2. The type of the field
3. The description of the field
4. The offset of the field, relative to the beginning of the class
5. The pointer to the metaclass

Listing 16 Setting MetaField information

```
new MetaField(“lumi”, “double”, “Instantaneous luminosity”,
                &((MCEvent*)0)->m_lumi, mc);
```

Inside the constructor of MetaField several functions will be called which set the corresponding information (see Listing 17).

Listing 17 Constructor of MetaField

```
MetaField(std::string name, std::string type, std::string desc
          void* offset, MetaClass* declaringClass)
{
    setName(name);
    setType(type);
    setDesc(desc);
    setOffset(offset);
    setDeclaringClass(declaringClass);
    declaringClass->addField(name, this);
}
```

A Methods

A.1 Methods for retrieving and manipulating of information

Table 1 Methods of class MetaMember

Methodname	***Returnvalue***	***Argument(s)***
name	string	-
desc	string	-
modifiers	int	-
declaringClass	MetaClass*	-

Table 2 Methods of class MetaClass

Methodname	***Returnvalue***	***Argument(s)***
field	MetaField*	string-
fields	vector<MetaField*>	-
method	MetaMethod*	string
methods	vector<MetaMethod*>	-
constructor	MetaConstructor*	string
constructors	vector<MetaConstructor*>	-
forName	MetaClass*	string
isPrimitive	bool	-
superClasses	vector<MetaClass*>	-

Table 3 Methods of class MetaField

Methodname	***Returnvalue***	***Argument(s)***
type	MetaClass*	-
toString	string	-
offset	int	-
getInt	int	void* base
getString	string	void* base

Table 3 Methods of class MetaField

type	MetaClass*	-
toString	string	-
offset	int	-
getFloat	float	void* base
getDouble	double	void* base
getBool	bool	void* base
getChar	char	void* base
getClass	int	void* base
setInt	void	void* base, int value
setString	void	void* base, string value
setFloat	void	void* base, float value
setDouble	void	void* base, double value
setBool	void	void* base, bool value
setChar	void	void*base, char value
setClass	void	void* base, int value

Table 4 Methods of class MetaModifier

MethodName	***Returntype***	***Argument(s)***
isPrivate	bool	int modifier
isProtected	bool	int modifier
isPublic	bool	int modifier
isConst	bool	int modifier
isVolatile	bool	int modifier
isAuto	bool	int modifier
isRegister	bool	int modifier
isStatic	bool	int modifier
isExtern	bool	int modifier
isMutable	bool	int modifier

Table 4 Methods of class MetaModifier

isInline	bool	int modifier
isVirtual	bool	int modifier
isExplicit	bool	int modifier
toString	string	int value

A.2 Methods for filling the MetaModel

Table 5 Methods of class MetaMember

Methodname	***Returnvalue***	***Argument(s)***
setName	void	string
setDesc	void	string
setModifiers	void	int
setDeclaringClass	void	MetaClass*

Table 6 Methods of class MetaClass

Methodname	***Returnvalue***	***Argument(s)***
initializePrimitives	void	-
addField	void	string-key, MetaField* value
addMethod	void	string key, MetaMethod* value
addConstructor	void	string key, MetaConstructor* value
addForName	void	string key, MetaClass* value
resolvePending	void	-
addPendingType	void	MetaField* key, string value
addPendingSuperClass	void	MetaClass* key, string value
addSuperClass	void	string name

Table 7 Methods of class MetaField

Methodname	***Returnvalue***	***Argument(s)***
setType	void	MetaClass* mcp
setType	void	string name
setOffset	void	void* offset

Table 8 Methods of class MetaModifier

MethodName	***Returntype***	***Argument(s)***
setPrivate	int	-
setProtected	int	-
setPublic	int	-
setConst	int	-
setVolatile	int	-
setAuto	int	-
setRegister	int	-
setStatic	int	-
setExtern	int	-
setMutable	int	-
setInline	int	-
setVirtual	int	-
setExplicit	int	-

B C++-code for filling the MetaModel

```
#include “GaudiKernel/Kernel.h”
#include <string>

#define private public
#include “../class1.h”
#undef private

#include “Reflection/Reflection.h”

class Class1_dict {
public: Class1_dict();
};

static Class1_dict instance;

Class1_dict::Class1_dict()
{
    MetaClass* metaC = new MetaClass(“Class1”, “This is the
        description for the first example class”);
    MetaClass::addForName(std::string(“Class1”), metaC);

    new MetaField(“varInt”, “int”, “Desc for int variable”,
        &((Class1*)0)->m_varInt, metaC);
    new MetaField(“varString”, “std::string”, “Desc for string var”,
        &((Class1*)0)->m_varString, metaC);
}
```