# The GEANT4 Experience —
# The Object Oriented Design and Production of a Large Program —
# CERN School of Computing, 1997

*John Allison*
The University of Manchester, M13 9PL, UK 9th October 1997

**Abstract**

GEANT4 is a detector simulation "toolkit", currently being implemented by an international group of about 100 physicists and programmers, under the auspices of CERN's Research and Development programme RD44. It is the first major piece of particle physics software to be written using object oriented techniques; the chosen language is C++. It will supersede GEANT3, a Fortran package which was becoming very difficult to maintain. GEANT4 will be much more flexible, and will allow the user to implement his or her own physics processes and geometrical shapes in an object oriented way. This paper describes our experience, half way into a 4 year project, of using these new methods and languages.

## 1 Introduction

The first "alpha release" of GEANT4 took place in April 1997. It was "consolidated" in July. It is an "at your risk" version, only available on request. Most of the basic physics processes are included and most of the GEANT3 shapes are represented.

Producing the alpha release has been a stressful time. Changes have been made right up to the last minute, and the alpha release is not bug free. In fact the April version was a little flaky, but the July version is usable.

I think this is typical of a software project. Remember, an alpha release is *not* a public release. Source can only be obtained by request, and our policy is only to release it to "friends", that is colleagues who understand the position and who will give constructive feedback which furthers the development of GEANT4.

What follows in Section 2 is our experience to date. Then, in Section 3 I will discuss some design issues, particularly those close to my own area of responsibility in GEANT4 and relevant to object oriented design.

## 2 The GEANT4 Experience

Most of us in the GEANT4 team came with little experience in object oriented methods (OOM). On the other hand, many had been involved in various aspects of particle physics computing for years and thus brought

a wealth of general experience and skills; quite a number had been involved in GEANT3. And we passionately believed that if GEANT was going to be useful in the LHC era, it had to be completely re-written. GEANT3 had become unmaintainable or, at least, undevelopable. And we believed the OO techniques were required.

Much of the first year of the project was devoted to design. We adopted the Booch OO design and analysis concepts and notation, with the help of Rational Rose design tool, but, from the first, it was recognised that it was also necessary to write prototype code in order to evolve the design (the philosophy of iterative evolution) and an early decision was made to use C++.

This turned out to be typical of the common sense decisions and philosophy simply "to make a start" and "to learn on the job", with help from attending courses, reading and simply using the knowledge of those with more experience. A requirements document was written (and published and publicised) — and the exercise helped clarify our objectives — but it contained little that was not "obvious" to most particle physicists, especially those with any experience in GEANT3. I doubt if many read it, once it had been written.

Thus began a very pragmatic distributed development process, with little formal analysis but with much trial and error (iterative evolution), many email exchanges, frequent meetings and a great deal of common sense, intelligence and good will. It has been an enjoyable experience. We have reached the "alpha tag" (a limited release) in about 2 years, roughly comparable to GEANT3 in functionality, and incorporating clever new geometry search algorithms, many improved physics processes and a more general tracking algorithm. This is quite an achievement. We have given ourselves another 2 years to produce a usable product.

What else do I want to say about GEANT4 development process?

- The project was divided into "components", "categories" or "domains" — digits+hits, events+tracks, g3tog4, geometry, global, graphics_reps, intercoms, interfaces, particle+matter, tracking, visualization — in total 1248 files and 209211 lines (a snapshot just before the alpha tag).
- Simple Gnu makefiles were made for each component and subcomponent, each including GNUmakefile.architecture and GNUmakefile.common. This has served its purpose but a more sophisticated build and release algorithm is needed.
- The code is maintained by CVS. CVS is more than a code repository; it is a tool for parallel code development, offering utilities for merging and versioning. It is indispensable.
- We have used and made extensive contributions to CLHEP, the C++ class library for particle physics.

- We use Rogue Wave tools.h++ template container classes (but will probably migrate to STL); such classes offer huge productivity gains and are very efficient.

The early releases were extremely immature, but informative enough for major experiments, such as BaBar, to plan for adoption. We hope that the alpha tag will be good enough to help people start coding their detector model and seeing some results (but, of course, it is still "at your risk").

Here are some thoughts on the product:

- GEANT4 is highly object oriented
- GEANT4 contains some highly sophisticated OO designs, particularly in geometry, tracking and intercoms
- GEANT4 contains a lot of physics, encapsulated in particle and process classes
- The abstract visualization interface seems to work
- The OO Method is essential
- Components developed more or less independently work together without serious conflicts/clashes/adverse interactions
- Abstract interfaces work
- Inheritance and code re-use works
- OOP leads to high number of small code units (functions)
- OOP leads to large number of files
- CVS is essential
- OO Design tools have been used successfully
- Performance is good

...and about people...

- ...thoughts about the GEANT4 team: in spite of initial inexperience on the part of most members (only one or two had previously been taught OO)...
  - All were experienced and established in computing
  - All were and are committed to OO
  - All were willing to put effort into learning OO
  - Most learnt "on the job"
  - All are now adequately proficient in C++ OO
- ...and thoughts about physicists in general
  - Physicists *can* learn OO Methods
  - There has to be a way of *easing* people into it
  - Physicists are good at learning on the job

- Needs dedication to learn full OO
- Can use "recipe" methods to begin with
- Start with "user defined types"
- Needs 2 years to grasp concepts of OO and adequate mastery of language!
- Need to acquire new way of thinking
- It just comes with practice — difficult to teach
- It comes more naturally to some than others
- Once acquired, one feels it is the "natural" way of thinking
- Young people will know nothing else
- It's fun

My conclusion? Using object oriented methods is definitely the way to go!

## 3 Some Design Issues in GEANT4

### 3.1 Decoupling the Components

Here is a fragment of the command line which makes a typical GEANT4 application which uses graphics.

```
... -lG4visualization -lG4ATLAS -lG4run -lG4interfaces -lG4event
-lG4intercoms -lG4tracking -lG4processes -lG4particles -lG4piiman
-lG4track -lG4materials -lG4particles -lG4piiman -lG4digits+hits
-lG4geometry -lG4graphics_reps -lG4global -lrwtool -L/usr/X11/lib
-lGLX -lGLU -lGL -lXm -lXpm -lXmu -lXt -lXext -lX11 -lm
```

It indicates the order of linking of libraries. We have been careful to avoid "circular dependencies"; even so, two libraries (`G4particles` and `G4piiman`) are included twice because we have not yet completely succeeded (it will be fixed!). (Most Unix systems search libraries only once, in the order specified, so you have to be careful about the order, and re-link a library if there are any circular dependencies.)

But what I want to draw your attention to is this: `G4visualization` is first! How can that be? Surely all other code uses visualization, not the other way round!

There are three answers:

1. Visualization *does* use other components, for example, geometry for drawing the detector, tracking for trajectories — in fact, I expect just about every component will be "used" by visualization by the time GEANT4 becomes a mature product. So it is right that the visualization library is first in the list.
2. Other components may use visualization but only through two abstract interfaces — `G4VVisManager` and `G4VGraphicsScene` — and these pure abstract interfaces (i.e., containing only pure

virtual functions) exist not in the visualization component but in graphics_reps, for convenience, which the last-but-one GEANT4 library to be linked. Thus the linker is satisfied.

3. Other components may use visualization through the command intercommunication system, just as though a command had been typed. The command intercommunication library is `G4intercoms`. It also contains abstract interfaces through which it communicates with other components.

This ability to define an abstract interface in C++ and other object-oriented languages is OO's most powerful feature. It decouples components from each other. It allows much more freedom of independent code development, and a much better guarantee that the components will inter-work satisfactorily when the code is brought together. It manages complexity, reduces re-compilation time drastically and can be used to eliminate circular dependencies between libraries.

### 3.2 Decoupling the Command Intercommunication Component

How does this work? Any component which wishes to respond to commands has to have a "messenger" which is an object of a class which inherits from the abstract interface `G4UImessenger`:

```
class G4UImessenger
{
  public:
      G4UImessenger();
      virtual ~G4UImessenger();
      virtual G4String getCurrentValue(G4UIcommand * command);
      virtual void setNewValue(G4UIcommand * command,G4String newValue);
  protected:
      void addUIcommand(G4UIcommand * newCommand);
};
```

(The virtual functions are not pure virtual, but they could be — they contain only safety code or no code. They are never called for any sensible concrete messenger.)

The concrete messenger registers commands and passes its own address (`this`):

```
  ...
  command = new G4UIcommand ("/vis/clear/view", this);
  command -> set_guidance ("Clears visible window of current view.");
  addUIcommand (command);
  ...
```

All the right calls are made through the virtual function mechanism with statements of the type:

```
  pMessenger -> setNewValue (pCommand, command_string);
```

The compiler does not need to know at compile time, nor the linker at link time, what the address of the concrete messenger is. Thus compilation dependencies are eliminated and unresolved references are not generated. The links are all made at run time.

This is the classical way of using abstract interfaces to decouple components. The GEANT4 command intercommunication system is like a "slow controls" system much loved by on-line system designers.

The GEANT4 command intercommunication system was designed and implemented by Makoto Asai of Hiroshima University.

### 3.3 Decoupling the Visualization Component

The visualization abstract interface is slightly different —

```
class G4VVisManager {
public:
  static G4VVisManager* GetConcreteInstance ();
  // Returns pointer to actual visualization manager if a view is
  // available for drawing, else returns null.  Always check value.
  ...
  ... plus many pure virtual methods, e.g., Draw (const G4Polyline&).
  ...
```

A static pointer is defined at load time to be zero. The above `Get` function returns this pointer. The real `G4VisManager` is a (singleton) object of a class which is publicly derived from `G4VVisManager` (note the extra `V`) and thus inherits the pointer, which it maintains; it is non-zero only when a valid view is available for drawing. Thus all code must test this pointer before using —

```
  G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
  if(pVVisManager) {
    G4Polyline pl;
    ...
    pVVisManager -> Draw (pl);
  }
```

Thus, kernel code appears to be using only the abstract base class, but actually calls the real `G4VisManager` through the virtual function mechanism. The real `G4VisManager` is thus decoupled from the kernel.

### 3.4 Describing Unknown Shapes to Unknown Scenes

Finally, I'll discuss a problem which is solved by the method known as "double dispatch". The ordinary use of virtual functions is equivalent to "single dispatch", that is, the resolving of function calls at run time. Double dispatch is simply two successive applications of the virtual function mechanism.

When a GEANT4 user requests that a detector volume is drawn, the Visualization Manager calls a function of the current view (by ordinary

single dispatch) which calls a function of the corresponding scene (an object of a class derived from G4VScene, itself derived from the abstract interface class G4VGraphicsScene), which obtains a G4VSolid* named, say, pSol. pSol points to an object — a G4Box or G4Tubs or whatever, objects of a class derived from G4VSolid. G4VScene does not know which particular shape of volume it is, but it can access functions of it through the virtual function mechanism, thus:

```
    pSol -> DescribeYourselfTo (*this);
```

This is the first step of the double dispatch. Notice it passes a reference to itself (it could as well have passed a pointer). Thus the function of the corresponding shape is called. If it was a G4Box:

```
void G4Box::DescribeYourselfTo (G4VGraphicsScene& scene) const {
  scene.AddThis (*this);
}
```

and if a G4Tubs:

```
void G4Tubs::DescribeYourselfTo (G4VGraphicsScene& scene) const {
  scene.AddThis (*this);
}
```

This is the second step of the double dispatch. It succeeds in calling the appropriate function — AddThis (const G4Box&) or AddThis (const G4Tubs&) or whatever — of the current scene — G4OpenGLScene or G4FukuiRendererScene or whatever. There is a matrix of possibilities.

All the required indexed jumps are put in place optimally by the compiler; not an if or a switch in sight! It is very fast.