Advanced European Infrastructures for Detectors at Accelerators

DDG4

A Simulation Toolkit for High Energy Physics Experiments using Geant4 and the DD4hep Geometry Description

M.Frank
CERN, 1211 Geneva 23, Switzerland

DDG4 User Manual

Abstract

Simulating the detector response is an essential tool in high energy physics to analyze the sensitivity of an experiment to the underlying physics. Such simulation tools require a detailed though convenient detector description as it is provided by the `DD4hep` toolkit. We will present the generic simulation toolkit `DDG4`using the `DD4hep` detector description toolkit. The toolkit implements a modular and flexible approach to simulation activities using Geant4. User defined simulation applications using `DDG4`can easily be configured, extended using specialzed action routines. The design is strongly driven by easy of use; developers of detector descriptions and applications using it them should provide minimal information and minimal specific code to achieve the desired result.

Document History		
Document version	**Date**	**Author**
1.0	**19/11/2013**	**Markus Frank CERN/LHCb**

Contents

1 Introduction

This manual should introduce to the DDG4 framework. One goal of DDG4is to easily configure the simulation applications capable of simulating the physics response of detector configurations as they are used for example in high energy physics experiments. In such simulation programs the user normally has to define to experimental setup in terms of its geometry and in terms of its active elements which sample the detector response.
The goal of DDG4is to generalize the configuration of a simulation application to a degree, which does not force users to write code to test a detector design. At the same time it should of course be feasible to supply specialized user written modules which are supposed to seamlessly operate together with standard modules supplied by the toolkit. Detector-simulation depends strongly on the use of an underlying simulation toolkit, the most prominent candidate nowadays being Geant4 [8]. DD4hep supports simulation activities with Geant4 providing an automatic translation mechanism between geometry representations. The simulation response in the active elements of the detector is strongly influenced by the technical choices and precise simulations depends on the very specific detection techniques.
Similar to the aim of DD4hep [1], where with time a standard palette of detector components developed by users should become part of the toolkit, DDG4also hopes to provide a standard palette of components used to support simulation activities for detector layouts where detector designers may base the simulation of a planned experiment on these predefined components for initial design and optimization studies.
This is not a manual to Geant4 nor the basic infrastructure of DD4hep . It is assumed that this knowledge is present and the typical glossary is known.

2 The Geant4 User Interface

The Geant4 simulation toolkit [8] implements a very complex machinery to simulate the energy deposition of particles traversing materials. To easy its usage for the clients and to shield clients from the complex internals when actually implementing a simulation applications for a given detector design, it provides several user hooks as shown in Figure 1. Each of these hooks serves a well specialized purpose, but unfortunately also leads to very specialized applications. One aim of DDG4is to formalize these user actions so that the invocation at the appropriate time may be purely data driven.
In detail the following object-hooks allow the client to define user provided actions:

- The **User Physics List** allows the client to customize and define the underlying physics process(es) which define the particle interactions inside the detector defined with the geometry description. These interactions define the detector response in terms of energy depositions.
- The **Run Action** is called once at the start and end of a run. i.e. a series of generated events. These two callbacks allow clients to define run-dependent actions such as statistics summaries etc.
- The **Primary Generator Action** is called for every event. During the callback all particles are created which form the microscopic kinematic action of the particle collision. This input may either origin directly from an event generator program or come from file.
- The **Event Action** is called once at the start and the end of each event. It is typically used for a simple analysis of the processed event. If the simulated data should be written to some persistent medium, the call at the end of the event processing is the appropriate place.
- The **Tracking Action**
- The **Stepping Action**
- The **Stacking Action**

Geant4 provides all callbacks with the necessary information in the form of appropriate arguments.
Besides the callback system, Geant4 provides callbacks whenever a particle traverses a sensitive volume. These callbacks are called - similar to event actions - once at the start and the end of the event, but

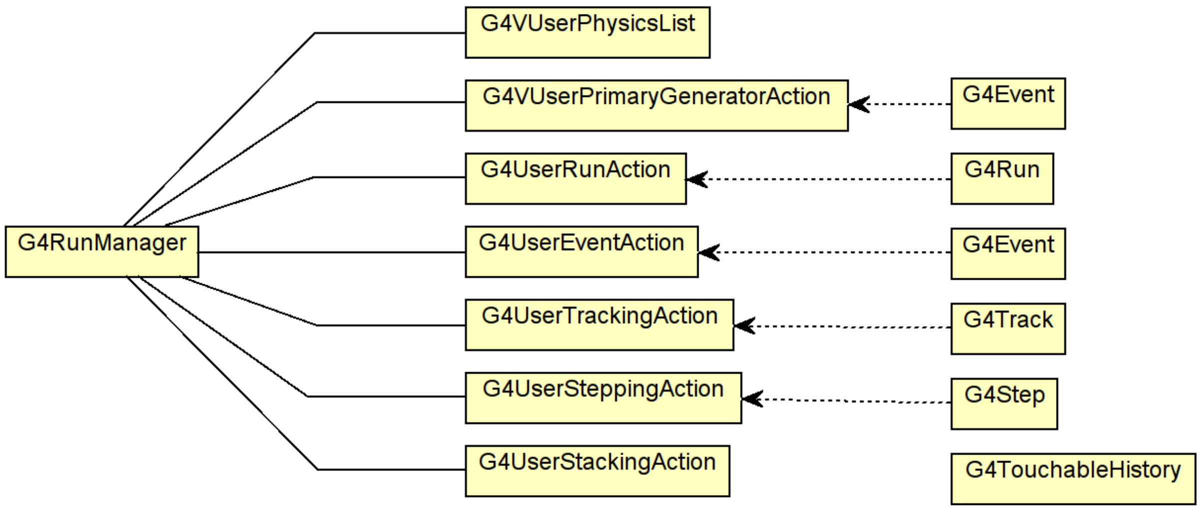

Figure 1: The various user hooks provided by Geant4. Not shown here is the callback system interfacing to the active elements of the detector design.

in addition, if either the energy deposit of a particle in the sensitive volume exceeds some threshold. The callbacks are formalized within the base class `G4VSensitiveDetector`.

3 DDG4 Implementation

A basic design criteria of the a `DDG4`simulation application was to process any user defined hook provided by Geant4 as a series of algorithmic procedures, which could be implemented either using inheritance or by a callback mechanism registering functions fulfilling a given signature. Such sequences are provided for all actions mentioned in the list in Section 2 as well as for the callbacks to sensitive detectors.
The callback mechanism was introduced to allow for weak coupling between the various actions. For example could an action performing monitoring using histograms at the event level initialize or reset its histograms at the start/end of each run. To do so, clearly a callback at the start/end of a run would be necessary.
In the following sections a flexible and extensible interface to hooks of Geant4 is discussed starting with the description of the basic components `Geant4Kernel` and `Geant4Action` followed by the implementation of the relevant specializations. The specializations exposed are sequences of such actions, which also call registered objects. In later section the configuration and the combination of these components forming a functional simulation application is presented.

3.1 The Application Core Object: `Geant4Kernel`

The kernel object is the central context of a `DDG4`simulation application and gives all clients access to the user hooks (see Figure 2). All Geant4 callback structures are exposed so that clients can easily objects implementing the required interface or register callbacks with the correct signature.

3.2 The Base Class of DDG4 Actions: `Geant4Action`

The class `Geant4Action` is a common component interface providing the basic interface to the framework to

- configure the component using a property mechanism

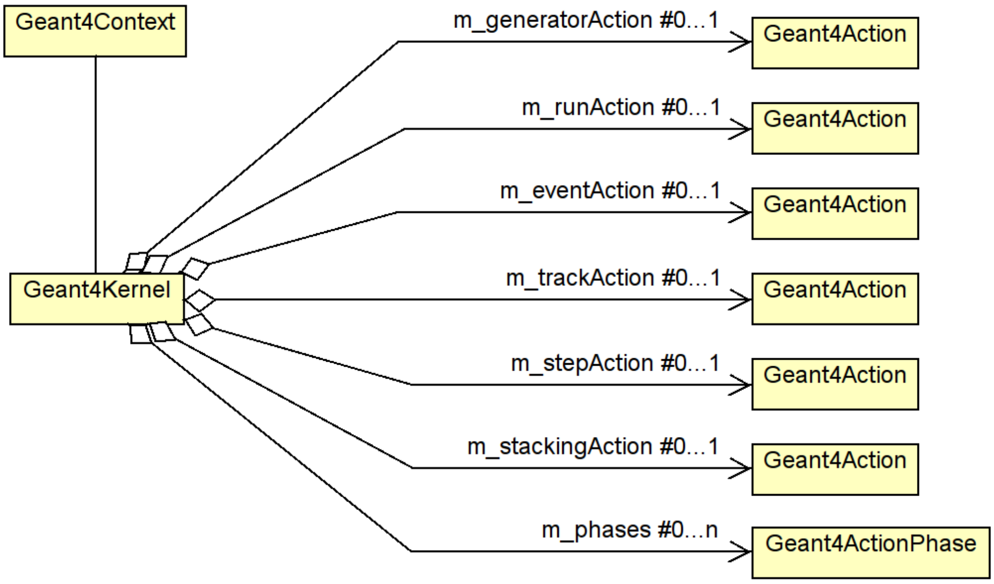

Figure 2: The sensitive detector design.

- provide an appropriate interface to Geant4 interactivity. The interactivity included a generic way to change and access properties from the Geant4 UI prompt as well as executing registered commands.
- As shown in Figure 3, the base class also provides to its sub-class a reference to the `Geant4Kernel` objects through the `Geant4Context`.

The `Geant4Action` is a named entity and can be uniquely identified within a sequence attached to one Geant4 user callback.

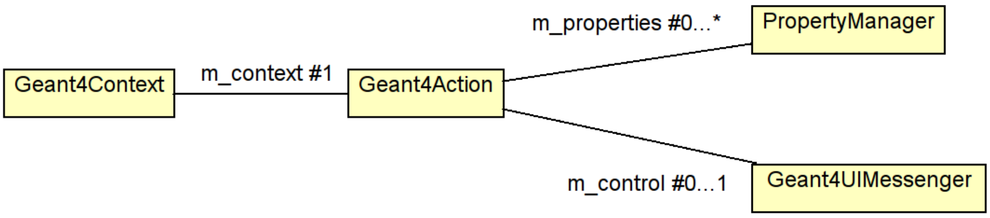

Figure 3: The design of the common base class `Geant4Action`.

`DDG4`knowns two types of actions: global actions and anonymouns actions. Global actions are accessible externally from the `Geant4Kernel` instance. Global actions are also re-usable and hence may be contribute to several action sequences (see the following chapters for details). Global actions are uniquely identified by their name. Anonymous actions are known only within one sequence and normally are not shared between sequences.

3.2.1 The Properties of Geant4Action Instances

Nearly any subclass of a `Geant4Action` needs a flexible configuration in order to be reused, modified etc. The implementation of the mechanism uses a very flexible value conversion mechanism using `boost::spirit`, which support also conversions between unrelated types provided a dictionary is present.

Properties are supposed to be member variables of a given action object. To publish a property it needs to be declared in the constructor as shown here:

```
declareProperty("OutputLevel", m_outputLevel = INFO);
declareProperty("Control",     m_needsControl = false);
```

The internal setup of the `Geant4Action` objects then ensure that all declared properties will be set after the object construction to the values set in the setup file.
Note: Because the values can only be set **after** the object was constructed, the actual values may not be used in the constructor of any base or sub-class.

3.3 Geant4 Action Sequences

The main action sequences have a fixed name. These are

- The **RunAction** attached to the `G4UserRunAction`, implemented by the `Geant4RunActionSequence` class and is called at the start and the end of every run (beamOn). Members of the `Geant4RunActionSequence` are of type `Geant4RunAction` and receive the callbacks by overloading the two routines:

  ```
  /// begin-of-run callback
  virtual void begin(const G4Run* run);
  /// End-of-run callback
  virtual void end(const G4Run* run);
  ```

  or register a callback with the signature `void (T::*)(const G4Run*)` either to receive begin-of-run or end-or-calls using the methods:

  ```
  /// Register begin-of-run callback. Types Q and T must be polymorph!
  template <typename Q, typename T> void callAtBegin(Q* p, void (T::*f)(const G4Run*));
  /// Register end-of-run callback. Types Q and T must be polymorph!
  template <typename Q, typename T> void callAtEnd(Q* p, void (T::*f)(const G4Run*));
  ```

  of the `Geant4RunActionSequence` from the `Geant4Context` object.

- The **EventAction** attached to the `G4UserEventAction`, implemented by the `EventActionSequence` class and is called at the start and the end of every event. Members of the `Geant4EventActionSequence` are of type `Geant4EventAction` and receive the callbacks by overloading the two routines:

  ```
  /// Begin-of-event callback
  virtual void begin(const G4Event* event);
  /// End-of-event callback
  virtual void end(const G4Event* event);
  ```

  or register a callback with the signature `void (T::*)(const G4Event*)` either to receive begin-of-run or end-or-calls using the methods:

  ```
  /// Register begin-of-event callback
  template <typename Q, typename T> void callAtBegin(Q* p, void (T::*f)(const G4Event*));
  /// Register end-of-event callback
  template <typename Q, typename T> void callAtEnd(Q* p, void (T::*f)(const G4Event*));
  ```

  of the `Geant4EventActionSequence` from the `Geant4Context` object.

- The **GeneratorAction** attached to the `G4VUserPrimaryGeneratorAction`, implemented by the `Geant4GeneratorActionSequence` class and is called at the start of every event and provided all initial tracks from the Monte-Carlo generator. Members of the `Geant4GeneratorActionSequence` are of type `Geant4EventAction` and receive the callbacks by overloading the member function:

  ```
  /// Callback to generate primary particles
  virtual void operator()(G4Event* event);
  ```

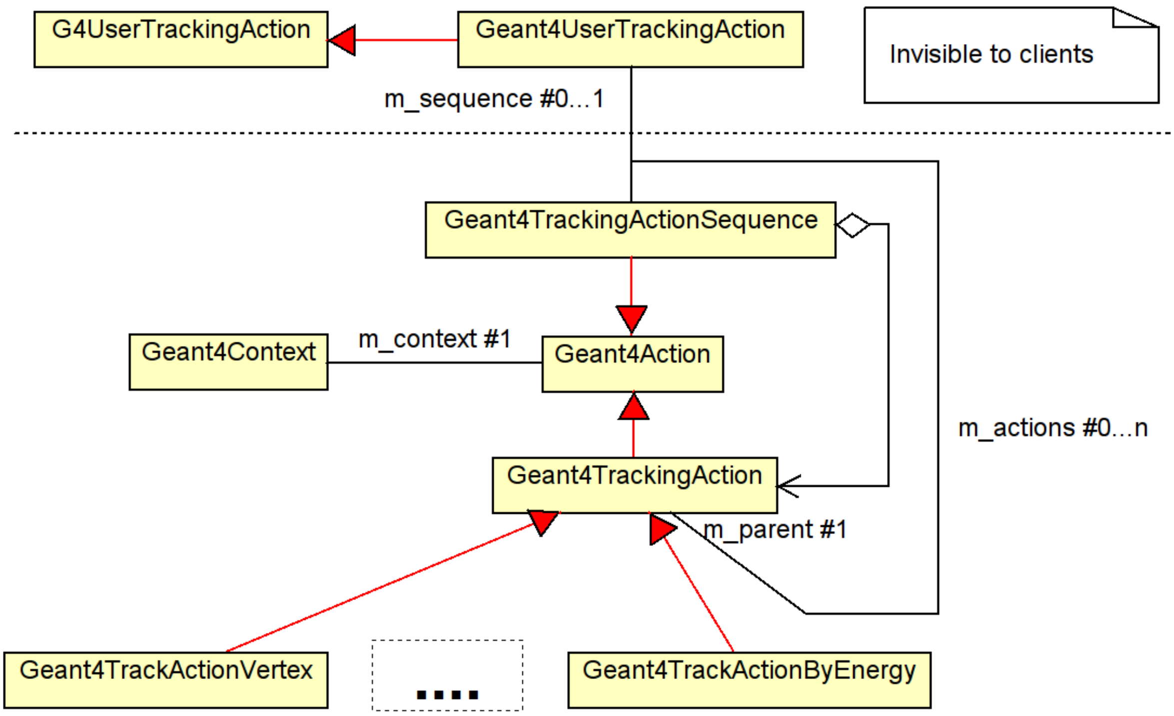

Figure 4: The design of the tracking action sequence. Specialized tracking action objects inherit from the `Geant4TrackingAction` object and must be attached to the sequence.

or register a callback with the signature `void (T::*)(G4Event*)` to receive calls using the method:

```
/// Register primary particle generation callback.
template <typename Q, typename T> void call(Q* p, void (T::*f)(G4Event*));
```

of the `Geant4GeneratorActionSequence` from the `Geant4Context` object.

- The **TrackingAction** attached to the `G4UserTrackingAction`, implemented by the `Geant4- Tracking-ActionSequence` class and is called at the start and the end of tracking one single particle trace through the material of the detector. Members of the `Geant4TrackingActionSequence` are of type `Geant4TrackingAction` and receive the callbacks by overloading the member function:

```
/// Pre-tracking action callback
virtual void begin(const G4Track* trk);
/// Post-tracking action callback
virtual void end(const G4Track* trk);
```

or register a callback with the signature `void (T::*)(const G4Step*, G4SteppingManager*)` to receive calls using the method:

```
/// Register Pre-track action callback
template <typename Q, typename T> void callAtBegin(Q* p, void (T::*f)(const G4Track*));
/// Register Post-track action callback
template <typename Q, typename T> void callAtEnd(Q* p, void (T::*f)(const G4Track*));
```

- The **SteppingAction** attached to the `G4UserSteppingAction`, implemented by the `Geant4- SteppingActionSequence` class and is called for each step when tracking a particle. Members of the `Geant4SteppingActionSequence` are of type `Geant4SteppingAction` and receive the callbacks by overloading the member function:

```
/// User stepping callback
virtual void operator()(const G4Step* step, G4SteppingManager* mgr);
```

or register a callback with the signature `void (T::*)(const G4Step*, G4SteppingManager*)` to receive calls using the method:

```
/// Register stepping action callback.
template <typename Q, typename T> void call(Q* p, void (T::*f)(const G4Step*,
                                                              G4SteppingManager*));
```

- The **StackingAction** attached to the `G4UserStackingAction`, implemented by the `Geant4StackingActionSequence` class. Members of the `Geant4StackingActionSequence` are of type `Geant4StackingAction` and receive the callbacks by overloading the member functions:

```
/// New-stage callback
virtual void newStage();
/// Preparation callback
virtual void prepare();
```

or register a callback with the signature `void (T::*)()` to receive calls using the method:

```
/// Register begin-of-event callback. Types Q and T must be polymorph!
template <typename T> void callAtNewStage(T* p, void (T::*f)());
/// Register end-of-event callback. Types Q and T must be polymorph!
template <typename T> void callAtPrepare(T* p, void (T::*f)());
```

All sequence types support the method `void adopt(T* member_reference)` to add the members. Once adopted, the sequence takes ownership and manages the member. The design of the sequences is very similar. Figure 4 show as an example the design of the `Geant4TrackingAction`.

3.4 Sensitive Detectors

Sensitive detectors are associated by the detector designers to all active materials, which would produce a signal which can be read out. In Geant4 this concept is realized by using a base class `G4VSensitiveDetector`, which receives a callback at the begin and the end of the event processing and at each step inside the active material whenever an energy deposition occurred.
The sensitive actions do not necessarily deal only the collection of energy deposits, but could also be used to simply monitor the performance of the active element e.g. by producing histograms of the absolute value or the spacial distribution of the depositions.
Within DDG4the concept of sensitive detectors is implemented as a configurable action sequence of type `Geant4SensDetActionSequence` calling members of the type `Geant4Sensitive` as shown in Figure 5. The actual processing part of such a sensitive action is only called if the and of a set of required filters of type `Geant4Filter` is positive (see also section **??**). No filter is also positive. Possible filters are e.g. particle filters, which ignore the sensitive detector action if the particle is a `geantino` or if the energy deposit is below a given threshold.
Objects of type `Geant4Sensitive` receive the callbacks by overloading the member function:

```
/// Method invoked at the begining of each event.
virtual void begin(G4HCofThisEvent* hce);
/// Method invoked at the end of each event.
virtual void end(G4HCofThisEvent* hce);
/// Method for generating hit(s) using the information of G4Step object.
virtual bool process(G4Step* step, G4TouchableHistory* history);
/// Method invoked if the event was aborted.
virtual void clear(G4HCofThisEvent* hce);
```

or register a callback with the signature `void (T::*)(G4HCofThisEvent*)` respectively `void (T::*)(G4Step*, G4TouchableHistory*)` to receive callbacks using the methods:

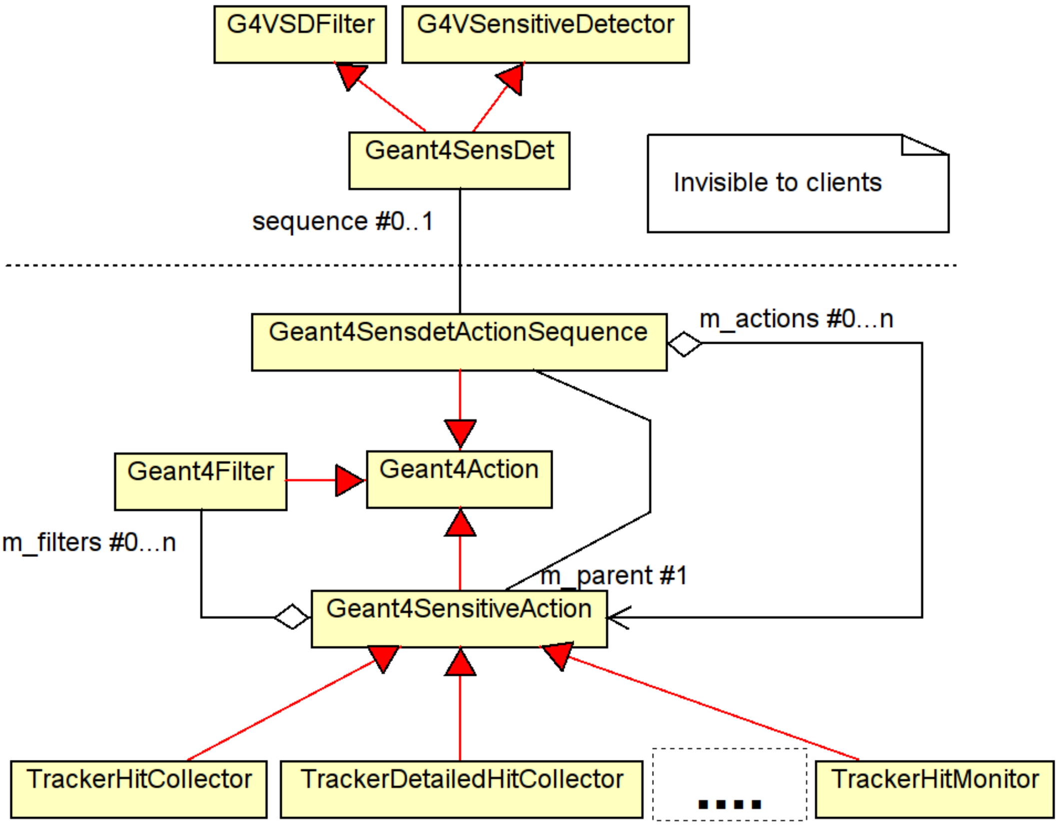

Figure 5: The sensitive detector design. The actual energy deposits are collected in user defined subclasses of the `Geant4Sensitive`. Here, as an example possible actions called `TrackerHitCollector`, `TrackerDetailedHitCollector` and `TrackerHitMonitor` are shown.

```
/// Register begin-of-event callback
template <typename T> void callAtBegin(T* p, void (T::*f)(G4HCofThisEvent*));
/// Register end-of-event callback
template <typename T> void callAtEnd(T* p, void (T::*f)(G4HCofThisEvent*));
/// Register process-hit callback
template <typename T> void callAtProcess(T* p, void (T::*f)(G4Step*, G4TouchableHistory*));
/// Register clear callback
template <typename T> void callAtClear(T* p, void (T::*f)(G4HCofThisEvent*));
```

3.4.1 Sensitive Detector Filters

Filters are called by Geant4 before the hit processing in the sensitive detectors start. The global filters may be shared between many sensitive detectors. Alternatively filters may be directly attached to the sensitive detector in question. Attributes are directly passed as properties to the filter action.

3.5 The Geant4 Physics List

Geant4 provides the base class `G4VUserPhysicsList`. Any user defined physics list must provide this interface. DDG4 provides such an interface through the ROOT plugin mechanism using the class `G4VModularPhysicsList`. The flexibility of DDG4allows for several possibilities to setup the Geant4 physics list.

- The **physics list** may be configured as a sequence of type `Geant4PhysicsListActionSequence` . Members of the `Geant4PhysicsListActionSequence` are of type `Geant4PhysicsList` and receive the callbacks by overloading the member functions:

  ```
  /// Callback to construct the physics constructors
  virtual void constructProcess(Geant4UserPhysics* interface);
  /// constructParticle callback
  virtual void constructParticles(Geant4UserPhysics* particle);
  /// constructPhysics callback
  virtual void constructPhysics(Geant4UserPhysics* physics);
  ```

  or register a callback with the signature `void (T::*)(Geant4UserPhysics*)` to receive calls using the method:

  ```
  /// Register process construction callback t
  template <typename Q, typename T> void constructProcess(Q* p, void (T::*f)(Geant4UserPhysics*));
  /// Register particle construction callback
  template <typename Q, typename T> void constructParticle(Q* p, void (T::*f)(Geant4UserPhysics*));
  ```

  The argument of type `Geant4UserPhysics` provides a basic interface to the original `G4VModularPhysicsList`, which allows to register physics constructors etc.

- In most of the cases the above approach is an overkill and often even too flexible. Hence, alternatively, the physics list may consist of a single entry of type `Geant4PhysicsList` .

The basic implementation of the `Geant4PhysicsList` supports the usage of various

- particle constructors , such as single particle constructors like `G4Gamma` or `G4Proton`, or whole particle groups like `G4BosonConstructor` or `G4IonConstrutor`,
- physics process constructors , such as e.g. `G4GammaConversion`, `G4PhotoElectricEffect` or `G4ComptonScattering`,
- physics constructors combining particles and the corresponding interactions, such as e.g. `G4OpticalPhysics`, `HadronPhysicsLHEP` or `G4HadronElasticPhysics` and
- predefined Geant4 physics lists , such as `FTFP_BERT`, `CHIPS` or `QGSP_INCLXX`. This option is triggered by the content of the string property "extends" of the `Geant4Kernel::physicsList()` action.

These constructors are internally connected to the above callbacks to register themselves. The constructors are instantiated using the ROOT plugin mechanism.
The description of the above interface is only for completeness. The basic idea is, that the physics list with its particle and physics constructors is configured entirely data driven using the setup mechanism described in the following chapter. However, DDG4 is not limited to the data driven approach. Specialized physics lists may be supplied, but there should be no need. New physics lists could always be composed by actually providing new physics constructors and actually publishing these using the factory methods:

```
// Framework include files
#include "DDG4/Factories.h"

#include "My_Very_Own_Physics_Constructor.h"
DECLARE_GEANT4_PHYSICS(My_Very_Own_Physics_Constructor)
```

where `My_Very_Own_Physics_Constructor` represents a sub-class of `G4VPhysicsConstructor`.

3.6 The Support of the Geant4 UI: `Geant4UIMessenger`

The support of interactive in Geant4 is absolutely mandatory to debug detector setups in small steps. The Geant4 toolkit did provide for this reason a machinery of UI commands.

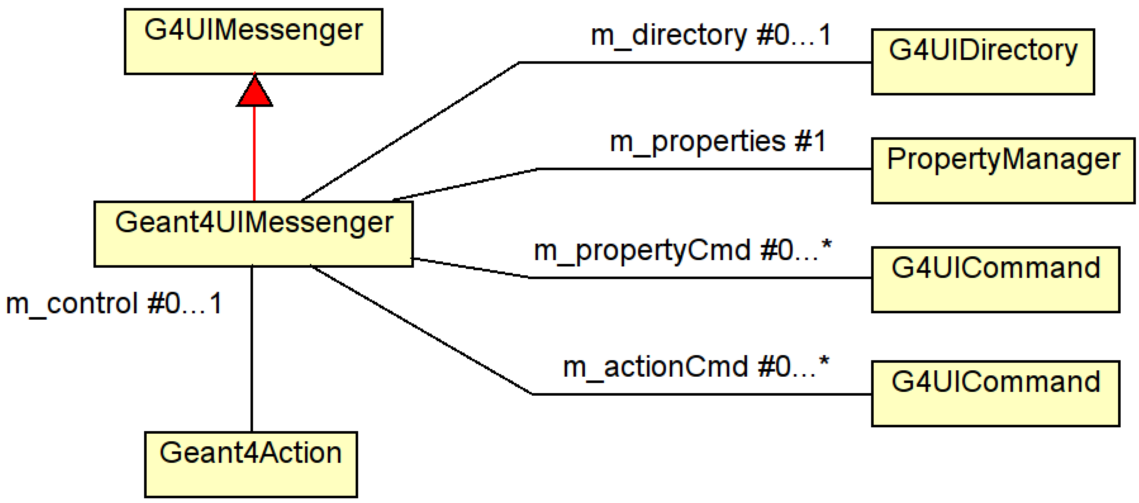

Figure 6: The design of the `Geant4UIMessenger` class responsible for the interaction between the user and the components of `DDG4`and Geant4.

The UI control is enabled, as soon as the property "Control" (boolean) is set to true. Be default all properties of the action are exported. Similar to the callback mechanism described above it is also feasible to register any object callback invoking a method of a `Geant4Action`-subclass.
The following (shortened) screen dump illustrates the usage of the generic interface any Geant4Action offers:

```
Idle> ls
Command directory path : /
 Sub-directories :
   /control/   UI control commands.
   /units/   Available units.
   /process/   Process Table control commands.
   /ddg4/   Control for all named Geant4 actions
   ...
Idle> cd /ddg4
Idle> ls
...
Control for all named Geant4 actions

 Sub-directories :
   /ddg4/EventAction/   Control hierarchy for Geant4 action:EventAction
   /ddg4/RunAction/   Control hierarchy for Geant4 action:RunAction
   /ddg4/Gun/   Control hierarchy for Geant4 action:Gun
   /ddg4/GeneratorAction/   Control hierarchy for Geant4 action:GeneratorAction
   /ddg4/PhysicsList/   Control hierarchy for Geant4 action:PhysicsList
Idle> ls Gun
...
Control hierarchy for Geant4 action:Gun

 Sub-directories :
```

```
 Commands :
   show * Show all properties of Geant4 component:Gun
...
   particle * Property item of type std::string
   pos_x * Property item of type double
   pos_y * Property item of type double
   pos_z * Property item of type double
Idle> Gun/show
...
PropertyManager: Property multiplicity = 1
PropertyManager: Property name = 'Gun'
PropertyManager: Property particle = 'e-'
PropertyManager: Property pos_x = 0
PropertyManager: Property pos_y = 0
PropertyManager: Property pos_z = 0

Idle> Gun/pos_z 1.0
Geant4UIMessenger: +++ Gun> Setting new property value pos_z = 1.0.
Idle> Gun/pos_y 1.0
Geant4UIMessenger: +++ Gun> Setting new property value pos_y = 1.0.
Idle> Gun/pos_x 1.0
Geant4UIMessenger: +++ Gun> Setting new property value pos_x = 1.0.
Idle> Gun/show
...
PropertyManager: Property pos_x = 1
PropertyManager: Property pos_y = 1
PropertyManager: Property pos_z = 1
```

4 Setting up DDG4

4.1 Setting up DDG4 using XML

A special plugin was developed to enable the configuration of DDG4using XML structures. These files are parsed identically to the geometry setup in DD4hep the only difference is the name of the root-element, which for DDG4is <geant4_setup>. The following code snippet shows the basic structure of a DDG4setup file:

```
<geant4_setup>
  <physicslist>           ,,,  </physicslist>  <!-- Defintiion of the physics list          -->
  <actions>               ...  </actions>      <!-- The list of global actions             -->
  <phases>                ...  </phases>       <!-- The definition of the various phases   -->
  <filters>               ...  </filters>      <!-- The list of global filter actions      -->
  <sequences>             ...  </sequences>    <!-- The list of defined sequences          -->
  <sensitive_detectors>   ...  </sensitive_detectors>  <!-- The list of sensitive detectors -->
  <properties>            ...  </properties>   <!-- Free format option sequences           -->
</geant4_setup>
```

To setup a DDG44 application any number of xml setup files may be interpreted iteratively. In the following subsections the content of these first level sub-trees will be discussed.

4.1.1 Setup of the Physics List

The main tag to setup a physics list is <physicslist> with the name attribute defining the instance of the Geant4PhysicsList object. An example code snippet is shown below in Figure 7.

```
<geant4_setup>
  <physicslist name="Geant4PhysicsList/MyPhysics.0">

    <extends name="QGSP_BERT"/>                    <!-- Geant4 basic Physics list -->

    <particles>                                    <!-- Particle constructors     -->
      <construct name="G4Geantino"/>
      <construct name="G4ChargedGeantino"/>
      <construct name="G4Electron"/>
      <construct name="G4Gamma"/>
      <construct name="G4BosonConstructor"/>
      <construct name="G4LeptonConstructor"/>
      <construct name="G4MesonConstructor"/>
      <construct name="G4BaryonConstructor"/>
      ...
    </particles>

    <processes>                                    <!-- Process constructors      -->
      <particle name="e[+-]" cut="1*mm">
        <process name="G4eMultipleScattering"  ordAtRestDoIt="-1"      ordAlongSteptDoIt="1"
                                               ordPostStepDoIt="1"/>
        <process name="G4eIonisation"          ordAtRestDoIt="-1"      ordAlongSteptDoIt="2"
                                               ordPostStepDoIt="2"/>
      </particle>
      <particle name="mu[+-]">
        <process name="G4MuMultipleScattering" ordAtRestDoIt="-1"      ordAlongSteptDoIt="1"
                                               ordPostStepDoIt="1"/>
        <process name="G4MuIonisation"         ordAtRestDoIt="-1"      ordAlongSteptDoIt="2"
                                               ordPostStepDoIt="2"/>
      </particle>
```

```
31       ...
32     </processes>
33
34     <physics>                                     <!-- Physics constructors      -->
35       <construct name="G4EmStandardPhysics"/>
36       <construct name="HadronPhysicsQGSP"/>
37       ...
38     </physics>
39
40   </physicslist>
41 </geant4_setup>
```

Figure 7: XML snippet showing the configuration of a physics list.

To trigger a call to a

- **particle constructors** (line 7-14), use the `<particles>` section and define the Geant4 particle constructor to be called by name. To trigger a call to
- **physics process constructors**, as shown in line 19-30, Define for each particle matching the name pattern (regular expression!) and the default cut value for the corresponding processes. The attributes ordXXXX correspond to the arguments of the Geant4 call `G4ProcessManager::AddProcess(process,ordAtRestDoIt, ordAlongSteptDoIt,ordPostStepDoIt);` The processes themself are created using the ROOT plugin mechanism. To trigger a call to
- **physics constructors**, as shown in line 34-35, use the `<physics>` section and
- to base all these constructs on an already existing predefined Geant4 physics list use the `<extends>` tag with the attribute containing the name of the physics list as shown in line 4.

If only a predefined physics list is used, which probably already satisfies very many use cases, all these section collapse to:

```
<geant4_setup>
  <physicslist name="Geant4PhysicsList/MyPhysics.0">
    <extends name="QGSP_BERT"/>                    <!-- Geant4 basic Physics list -->
  </physicslist>
</geant4_setup>
```

4.1.2 Setup of Global Geant4 Actions

Global actions must be defined in the `<actions>` section as shown in the following snippet:

```
<geant4_setup>
  <actions>
    <action name="Geant4TestRunAction/RunInit">
      <properties Property_int="12345"
          Property_double="-5e15"
          Property_string="Startrun: Hello_2"/>
     </action>
    <action name="Geant4TestEventAction/UserEvent_2"
            Property_int="1234"
            Property_double="5e15"
            Property_string="Hello_2" />
  </actions>
</geant4_setup>
```

The default properties of **every** `Geant4Action` object are:

```
Name        [string]              Action name
OutputLevel [int]                 Flag to customize the level of printout
Control     [boolean]             Flag if the UI messenger should be installed.
```

The `name` attribute of an action child is a qualified name: The first part denotes the type of the plugin (i.e. its class), the second part the name of the instance. Within one collection the instance `name` must be unique. Properties of Geant4Actions are set by placing them as attributes into the `<properties>` section.

4.1.3 Setup of Geant4 Filters

Filters are special actions called by `Geant4Sensitives`. Filters may be global or anonymous i.e. reusable by several sensitive detector sequences as illustrated in Section 4.1.4. The setup is analogous to the setup of global actions:

```
<filters>
  <filter name="GeantinoRejectFilter/GeantinoRejector"/>
  <filter name="ParticleRejectFilter/OpticalPhotonRejector">
    <properties particle="opticalphoton"/>
  </filter>
  <filter name="ParticleSelectFilter/OpticalPhotonSelector">
    <properties particle="opticalphoton"/>
  </filter>
  <filter name="EnergyDepositMinimumCut">
    <properties Cut="10*MeV"/>
  </filter>
  <!--         ... next global filter ...       -->
</filters>
```

Global filters are accessible from the `Geant4Kernel` object.

4.1.4 Geant4 Action Sequences

`Geant4 Action Sequences` by definition are `Geant4Action` objects. Hence, they share the setup mechanism with properties etc. For the setup mechanism two different types of sequences are known to `DDG4`: *Action sequences* and *Sensitive detector sequences*. Bot are declared in the `sequences` section:

```
<geant4_setup>
  <sequences>
    <sequence name="Geant4EventActionSequence/EventAction"> <!-- Sequence "EventAction" of type
                                                                 "Geant4EventActionSequence" -->
      <action name="Geant4TestEventAction/UserEvent_1">     <!-- Anonymouns action             -->
        <properties Property_int="01234"                    <!-- Properties go inline          -->
            Property_double="1e11"
            Property_string="'Hello_1'"/>
      </action>
      <action name="UserEvent_2"/>                          <!-- Global action defined in "actions" -->
                                                            <!-- Only the name is referenced here   -->
      <action name="Geant4Output2ROOT/RootOutput">          <!-- ROOT I/O action               -->
        <properties Output="simple.root"/>                  <!-- Output file property          -->
      </action>
      <action name="Geant4Output2LCIO/LCIOOutput">          <!-- LCIO output action            -->
        <properties Output="simple_lcio"/>                  <!-- Output file property          -->
      </action>
    </sequence>


    <sequence sd="SiTrackerBarrel" type="Geant4SensDetActionSequence">
```

```
      <filter name="GeantinoRejector"/>
      <filter name="EnergyDepositMinimumCut"/>
      <action name="Geant4SimpleTrackerAction/SiTrackerBarrelHandler"/>
    </sequence>
    <sequence sd="SiTrackerEndcap" type="Geant4SensDetActionSequence">
      <filter name="GeantinoRejector"/>
      <filter name="EnergyDepositMinimumCut"/>
      <action name="Geant4SimpleTrackerAction/SiTrackerEndcapHandler"/>
    </sequence>
    <!--    ... next sequence ...     -->
  </sequences>
</geant4_setup>
```

Here firstly the **EventAction** sequence is defined with its members. Secondly a sensitve detector sequence is defined for the subdetector `SiTrackerBarrel` of type `Geant4SensDetActionSequence`. The sequence uses two filters: `GeantinoRejector` to not generate hits from geantinos and `EnergyDepositMinimumCut` to enforce a minimal energy deposit. These filters are global i.e. they may be applied by many subdetectors. The setup of global filters is described in Section 4.1.3. Finally the action `SiTrackerEndcapHandler` of type `Geant4SimpleTrackerAction` is chained, which collects the deposited energy and creates a collection of hits. The `Geant4SimpleTrackerAction` is a template callback to illustrate the usage of sensitive elements in `DDG4`. The resulting hit collection of these handlers by default have the same name as the object instance name. Analogous below the sensitive detector sequence for the subdetector `SiTrackerEndcap` is shown, which reuses the same filter actions, but will build its own hit collection.
Plase note:

- **It was already mentioned, but once again**: Event-, run-, generator-, tracking-, stepping- and stacking actions sequences have predefined names! These names are fixed and part of the **common knowledge**, they cannot be altered. Please refer to Section 3.3 for the names of the global action sequences.
- the sensitive detector sequences are matched by the attribute `sd` to the subdetectors created with the `DD4hep` detector description package. Values must match!
- In the event that several xml files are parsed it is absolutely vital that the `<actions>` section is interpreted **before** the `sequences`.
- For each XML file several `<sequences>` are allowed.

4.1.5 Setup of Geant4 Sensitive Detectors

```
<geant4_setup>
  <sensitive_detectors>
    <sd name="SiTrackerBarrel"
        type="Geant4SensDet"
        ecut="10.0*MeV"
        verbose="true"
        hit_aggregation="position">
    </sd>
    <!-- ...  next sensitive detector ... -->
  </sensitive_detectors>
</geant4_setup>
```

4.1.6 Miscellaneous Setup of Geant4 Objects

This section is used for the flexible setup of auxiliary objects such as the electromagnetic fields used in Geant4:

```
<geant4_setup>
```

```
  <properties>
    <attributes name="geant4_field"
          id="0"
          type="Geant4FieldSetup"
          object="GlobalSolenoid"
          global="true"
          min_chord_step="0.01*mm"
          delta_chord="0.25*mm"
          delta_intersection="1e-05*mm"
          delta_one_step="0.001*mm"
          eps_min="5e-05*mm"
          eps_max="0.001*mm"
          stepper="HelixSimpleRunge"
          equation="Mag_UsualEqRhs">
    </attributes>
    ...
  </properties>
</geant4_setup>
```

Important are the tags `type` and `object`, which are used to firstly define the plugin to be called and secondly define the object from the `DD4hep` description to be configured for the use within Geant4.

4.1.7 Setup of Geant4 Phases

Phases are configured as shown below. However, the use is **discouraged**, since it is not yet clear if there are appropriate use cases!

```
<phases>
  <phase type="RunAction/begin">
    <action name="RunInit"/>
    <action name="Geant4TestRunAction/UserRunInit">
  <properties Property_int="1234"
          Property_double="5e15"
          Property_string="'Hello_2'"/>
    </action>
  </phase>
  <phase type="EventAction/begin">
    <action name="UserEvent_2"/>
  </phase>
  <phase type="EventAction/end">
    <action name="UserEvent_2"/>
  </phase>
  ...
</phases>
```

4.2 Setting up DDG4 using ROOT-CINT

The setup of DDG4directly from the the ROOT interpreter using the AClick mechanism is very simple, but mainly meant for purists (like me ;-)), since it is nearly equivalent to the explicit setup within a C++ main program. The following code section shows how to do it. For explanation the code secment is discussed below line by line.

```
#include "DDG4/Geant4Config.h"
#include "DDG4/Geant4TestActions.h"
#include "DDG4/Geant4TrackHandler.h"
#include <iostream>

using namespace std;
using namespace DD4hep;
using namespace DD4hep::Simulation;
using namespace DD4hep::Simulation::Test;
using namespace DD4hep::Simulation::Setup;

#if defined(__MAKECINT__)
#pragma link C++ class Geant4RunActionSequence;
#pragma link C++ class Geant4EventActionSequence;
#pragma link C++ class Geant4SteppingActionSequence;
#pragma link C++ class Geant4StackingActionSequence;
#pragma link C++ class Geant4GeneratorActionSequence;
#pragma link C++ class Geant4Action;
#pragma link C++ class Geant4Kernel;
#endif

SensitiveSeq::handled_type* setupDetector(Kernel& kernel, const std::string& name)   {
  SensitiveSeq sd = SensitiveSeq(kernel,name);
  Sensitive  sens = Sensitive(kernel,"Geant4TestSensitive/"+name+"Handler",name);
  sd->adopt(sens);
  sens = Sensitive(kernel,"Geant4TestSensitive/"+name+"Monitor",name);
  sd->adopt(sens);
  return sd;
}

void exampleAClick()  {
  Geant4Kernel& kernel = Geant4Kernel::instance(LCDD::getInstance());
  kernel.loadGeometry("file:../DD4hep.trunk/DDExamples/CLICSiD/compact/compact.xml");
  kernel.loadXML("DDG4_field.xml");

  GenAction gun(kernel,"Geant4ParticleGun/Gun");
  gun["energy"] = 0.5*GeV;                          // Set properties
  gun["particle"] = "e-";
  gun["multiplicity"] = 1;
  kernel.generatorAction().adopt(gun);

  Action run_init(kernel,"Geant4TestRunAction/RunInit");
  run_init["Property_int"] = 12345;
  kernel.runAction().callAtBegin  (run_init.get(),&Geant4TestRunAction::begin);
  kernel.eventAction().callAtBegin(run_init.get(),&Geant4TestRunAction::beginEvent);
  kernel.eventAction().callAtEnd  (run_init.get(),&Geant4TestRunAction::endEvent);

  Action evt_1(kernel,"Geant4TestEventAction/UserEvent_1");
  evt_1["Property_int"] = 12345;                    // Set properties
  evt_1["Property_string"] = "Events";
  kernel.eventAction().adopt(evt_1);
```

```
52
53  EventAction evt_2(kernel,"Geant4TestEventAction/UserEvent_2");
54  kernel.eventAction().adopt(evt_2);
55
56  kernel.runAction().callAtBegin(evt_2.get(),&Geant4TestEventAction::begin);
57  kernel.runAction().callAtEnd  (evt_2.get(),&Geant4TestEventAction::end);
58
59  setupDetector(kernel,"SiVertexBarrel");
60  setupDetector(kernel,"SiVertexEndcap");
61  // .... more subdetectors here .....
62  setupDetector(kernel,"LumiCal");
63  setupDetector(kernel,"BeamCal");
64
65  kernel.configure();
66  kernel.initialize();
67  kernel.run();
68  std::cout << "Successfully executed application .... " << std::endl;
69  kernel.terminate();
70 }
```

Line	
1	The header file `Geant4Config.h` contains a set of wrapper classes to easy the creation of objects using the plugin mechanism and setting properties to `Geant4Action` objects. These helpers and the corresponding functionality are not included in the wrapped classes themselves to not clutter the code with stuff only used for the setup. All contained objects are in the namespace `DD4hep::Simulation::Setup`
. **6-10**	Save yourself specifying all the namespaces objects are in....
13-19	CINT processing pragmas. Classes defined here will be available at the ROOT prompt after this AClick is loaded.
22-29	Sampler to fill the sensitive detector sequences for each subdetector with two entries: a handler and a monitor action. Please note, that this here is example code and in real life specialized actions will have to be provided for each subdetector.
31	Let's go for it. here the entry point starts....
32	Create the `Geant4Kernel` object.
33	Load the geometry into `DD4hep` .
34	Redefine the setup of the sensitive detectors.
36-40	Create the generator action of type `Geant4ParticleGun` with name `Gun`, set non-default properties and activate the configured object by attaching it to the `Geant4Kernel`.
42-46	Create a user defined begin-of-run action callback, set the properties and attach it to the begin of run calls. To collect statistics extra member functions are registered to be called at the beginning and the end of each event.
48-51	Create a user defined event action routine, set its properties and attach it to the event action sequence.
53-54	Create a second event action and register it to the event action sequence. This action will be called after the previously created action.
56-57	For this event action we want to receive callbacks at start- and end-of-run to produce additional summary output.
59-63	Call the sampler routine to attach test actions to the subdetectors defined.
65-66	Configure, initialize and run the Geant4 application. Most of the Geant4 actions will only be created here and the action sequences created before will be attached now.
69	Terminate the Geant4 application and exit.

CINT currently cannot handle pointers to member functions [1]. Hence the above AClick only works in compiled mode. To invoke the compilation the following action is necessary from the ROOT prompt:

```
$> root.exe
  *******************************************
  *                                         *
  *        W E L C O M E  to  R O O T       *
  *                                         *
  *   Version   5.34/10    29 August 2013   *
  *                                         *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch            *
  *                                         *
  *******************************************

ROOT 5.34/10 (heads/v5-34-00-patches@v5-34-10-5-g0e8bac8, Sep 04 2013, 11:52:19 on linux)

CINT/ROOT C/C++ Interpreter version 5.18.00, July 2, 2010
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] .X initAClick.C
.... Setting up the CINT include pathes and the link statements.

root [1] .L ../DD4hep.trunk/DDG4/examples/exampleAClick.C+
Info in <TUnixSystem::ACLiC>: creating shared library ....exampleAClick_C.so
.... some Cint warnings concerning member function pointers .....

root [2] exampleAClick()
.... and it starts ...
```

The above scripts are present in the DDG4/example directory located in svn. The intialization script `initAClick.C` may require customization to cope with the installation pathes.

4.3 Setting up DDG4 using Python

Given the reflection interface of ROOT, the setup of the simulation interface using DD4hep is of course also possible using the python interpreted language. In the following code example the setup of Geant4 using the `ClicSid` example is shown using python [2].

```
#
#
import DDG4
from SystemOfUnits import *
#
#
"""

   DD4hep example setup using the python configuration

   @author  M.Frank
   @version 1.0

"""
def run():
  kernel = DDG4.Kernel()
```

[1]This may change in the future once ROOT uses `clang` and `cling` as the interpreting engine.
[2]For comparison, the same example was used to illustrate the setup using XML files.

```
kernel.loadGeometry("file:../DD4hep.trunk/DDExamples/CLICSiD/compact/compact.xml")
kernel.loadXML("DDG4_field.xml")

lcdd = kernel.lcdd()
print '+++   List of sensitive detectors:'
for i in lcdd.detectors():
  o = DDG4.DetElement(i.second)
  sd = lcdd.sensitiveDetector(o.name())
  if sd.isValid():
    print '+++  %-32s type:%s'%(o.name(), sd.type(), )

# Configure Run actions
run1 = DDG4.RunAction(kernel,'Geant4TestRunAction/RunInit')
run1.Property_int    = 12345
run1.Property_double = -5e15*keV
run1.Property_string = 'Startrun: Hello_2'
print run1.Property_string, run1.Property_double, run1.Property_int
run1.enableUI()
kernel.registerGlobalAction(run1)
kernel.runAction().add(run1)

# Configure Event actions
evt2 = DDG4.EventAction(kernel,'Geant4TestEventAction/UserEvent_2')
evt2.Property_int    = 123454321
evt2.Property_double = 5e15*GeV
evt2.Property_string = 'Hello_2 from the python setup'
evt2.enableUI()
kernel.registerGlobalAction(evt2)

evt1 = DDG4.EventAction(kernel,'Geant4TestEventAction/UserEvent_1')
evt1.Property_int=01234
evt1.Property_double=1e11
evt1.Property_string='Hello_1'
evt1.enableUI()

kernel.eventAction().add(evt1)
kernel.eventAction().add(evt2)

# Configure I/O
evt_root = DDG4.EventAction(kernel,'Geant4Output2ROOT/RootOutput')
evt_root.Control = True
evt_root.Output = "simple.root"
evt_root.enableUI()

evt_lcio = DDG4.EventAction(kernel,'Geant4Output2LCIO/LcioOutput')
evt_lcio.Output = "simple_lcio"
evt_lcio.enableUI()

kernel.eventAction().add(evt_root)
kernel.eventAction().add(evt_lcio)

# Setup particle gun
gun = DDG4.GeneratorAction(kernel,"Geant4ParticleGun/Gun")
gun.energy   = 0.5*GeV
gun.particle = 'e-'
gun.multiplicity = 1
gun.enableUI()
kernel.generatorAction().add(gun)
```

```
# Setup global filters fur use in sensntive detectors
f1 = DDG4.Filter(kernel,'GeantinoRejectFilter/GeantinoRejector')
f2 = DDG4.Filter(kernel,'ParticleRejectFilter/OpticalPhotonRejector')
f2.particle = 'opticalphoton'
f3 = DDG4.Filter(kernel,'ParticleSelectFilter/OpticalPhotonSelector')
f3.particle = 'opticalphoton'
f4 = DDG4.Filter(kernel,'EnergyDepositMinimumCut')
f4.Cut = 10*MeV
f4.enableUI()
kernel.registerGlobalFilter(f1)
kernel.registerGlobalFilter(f2)
kernel.registerGlobalFilter(f3)
kernel.registerGlobalFilter(f4)

# First the tracking detectors
seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiVertexBarrel')
act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiVertexBarrelHandler','SiVertexBarrel')
seq.add(act)
seq.add(f1)
seq.add(f4)
act.add(f1)

seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiVertexEndcap')
act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiVertexEndcapHandler','SiVertexEndcap')
seq.add(act)
seq.add(f1)
seq.add(f4)

seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiTrackerBarrel')
act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiTrackerBarrelHandler','SiTrackerBarrel')
seq.add(act)
seq.add(f1)
seq.add(f4)

seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiTrackerEndcap')
act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiTrackerEndcapHandler','SiTrackerEndcap')
seq.add(act)

seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiTrackerForward')
act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiTrackerForwardHandler','SiTrackerForward')
seq.add(act)

# Now the calorimeters
seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/EcalBarrel')
act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/EcalBarrelHandler','EcalBarrel')
seq.add(act)

seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/EcalEndcap')
act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/EcalEndCapHandler','EcalEndcap')
seq.add(act)

seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/HcalBarrel')
act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/HcalBarrelHandler','HcalBarrel')
act.adoptFilter(kernel.globalFilter('OpticalPhotonRejector'))
seq.add(act)

act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/HcalOpticalBarrelHandler','HcalBarrel')
```

```
  act.adoptFilter(kernel.globalFilter('OpticalPhotonSelector'))
  seq.add(act)

  seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/HcalEndcap')
  act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/HcalEndcapHandler','HcalEndcap')
  seq.add(act)

  seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/HcalPlug')
  act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/HcalPlugHandler','HcalPlug')
  seq.add(act)

  seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/MuonBarrel')
  act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/MuonBarrelHandler','MuonBarrel')
  seq.add(act)

  seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/MuonEndcap')
  act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/MuonEndcapHandler','MuonEndcap')
  seq.add(act)

  seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/LumiCal')
  act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/LumiCalHandler','LumiCal')
  seq.add(act)

  seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/BeamCal')
  act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/BeamCalHandler','BeamCal')
  seq.add(act)

  # Now build the physics list:
  phys = kernel.physicsList()
  phys.extends = 'FTFP_BERT'
  #phys.transportation = True
  phys.decays  = True
  phys.enableUI()

  ph = DDG4.PhysicsList(kernel,'Geant4PhysicsList/Myphysics')
  ph.addParticleConstructor('G4BosonConstructor')
  ph.addParticleConstructor('G4LeptonConstructor')
  ph.addParticleProcess('e[+-]','G4eMultipleScattering',-1,1,1)
  ph.addPhysicsConstructor('G4OpticalPhysics')
  ph.enableUI()
  phys.add(ph)

  phys.dump()

  kernel.configure()
  kernel.initialize()
  kernel.run()
  kernel.terminate()

if __name__ == "__main__":
  run()
```

References

[1] DD4Hep web page, http://aidasoft.web.cern.ch/DD4hep.

[2] LHCb Collaboration, "LHCb, the Large Hadron Collider beauty experiment, reoptimised detector design and performance", CERN/LHCC 2003-030

[3] S. Ponce et al., "Detector Description Framework in LHCb", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2003), La Jolla, CA, 2003, proceedings.

[4] The ILD Concept Group, "The International Large Detector: Letter of Intent", ISBN 978-3-935702-42-3, 2009.

[5] H. Aihara, P. Burrows, M. Oreglia (Editors), "SiD Letter of Intent", arXiv:0911.0006, 2009.

[6] R.Brun, A.Gheata, M.Gheata, "The ROOT geometry package", Nuclear Instruments and Methods **A** 502 (2003) 676-680.

[7] R.Brun et al., "Root - An object oriented data analysis framework", Nuclear Instruments and Methods **A** 389 (1997) 8186.

[8] S. Agostinelli et al., "Geant4 - A Simulation Toolkit", Nuclear Instruments and Methods **A** 506 (2003) 250-303.

[9] T.Johnson et al., "LCGO - geometry description for ILC detectors", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2007), Victoria, BC, Canada, 2012, Proceedings.

[10] N.Graf et al., "lcsim: An integrated detector simulation, reconstruction and analysis environment", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2012), New York, 2012, Proceedings.

[11] R. Chytracek et al., "Geometry Description Markup Language for Physics Simulation and Analysis Applications", IEEE Trans. Nucl. Sci., Vol. 53, Issue: 5, Part 2, 2892-2896, http://gdml.web.cern.ch.

[12] C.Grefe et al., "The DDSegmentation package", Non existing documentation to be written.