Simulation — Geant4

HSF India HEP Software Workshop
December 20, 2023

Sunanda Banerjee

Simulation

- Detectors in nuclear and high-energy physics are getting more and more complex
  - carrying out an experiment with such a complex detector and interpreting the outcome needs some additional help
    - do a mock experiment on the computer with known sources and some model of the detector and see if the final results match the source
    - also use such mock experiments to design the detector to be used, in particular for very expensive experiments (space science, collider physics)
- Radiation is used in many other applications: medical diagnosis, medical treatment, non-destructive ways to probe structures
  - exposing living beings to radiation requires careful evaluation of the amount of radiation to be used
- Use the Monte Carlo technique to study these phenomena and Geant4 is a toolkit to enhance such a study

Geant4

- Geant4 is a toolkit which helps to build an application program for simulating the performance of a detector exposed to radiation
  - Originally built for experiments in high energy and nuclear physics
  - Also finds its application in space physics and medical science
- The public production version has been available since the year 1999
- The package came with several examples to guide buildings of application programs
- It is written mostly in the C++ language using object-oriented technology
- Available as open source through CERN and works on many platforms
  - Latest version Geant4.11.2
- Three main reference papers:
  - Nuclear Instruments and Methods A506 (2003) 250
  - IEEE Transactions on Nuclear Science 53 (2006) 270
  - Nuclear Instruments and Methods A835 (2016) 186

Detector Simulation

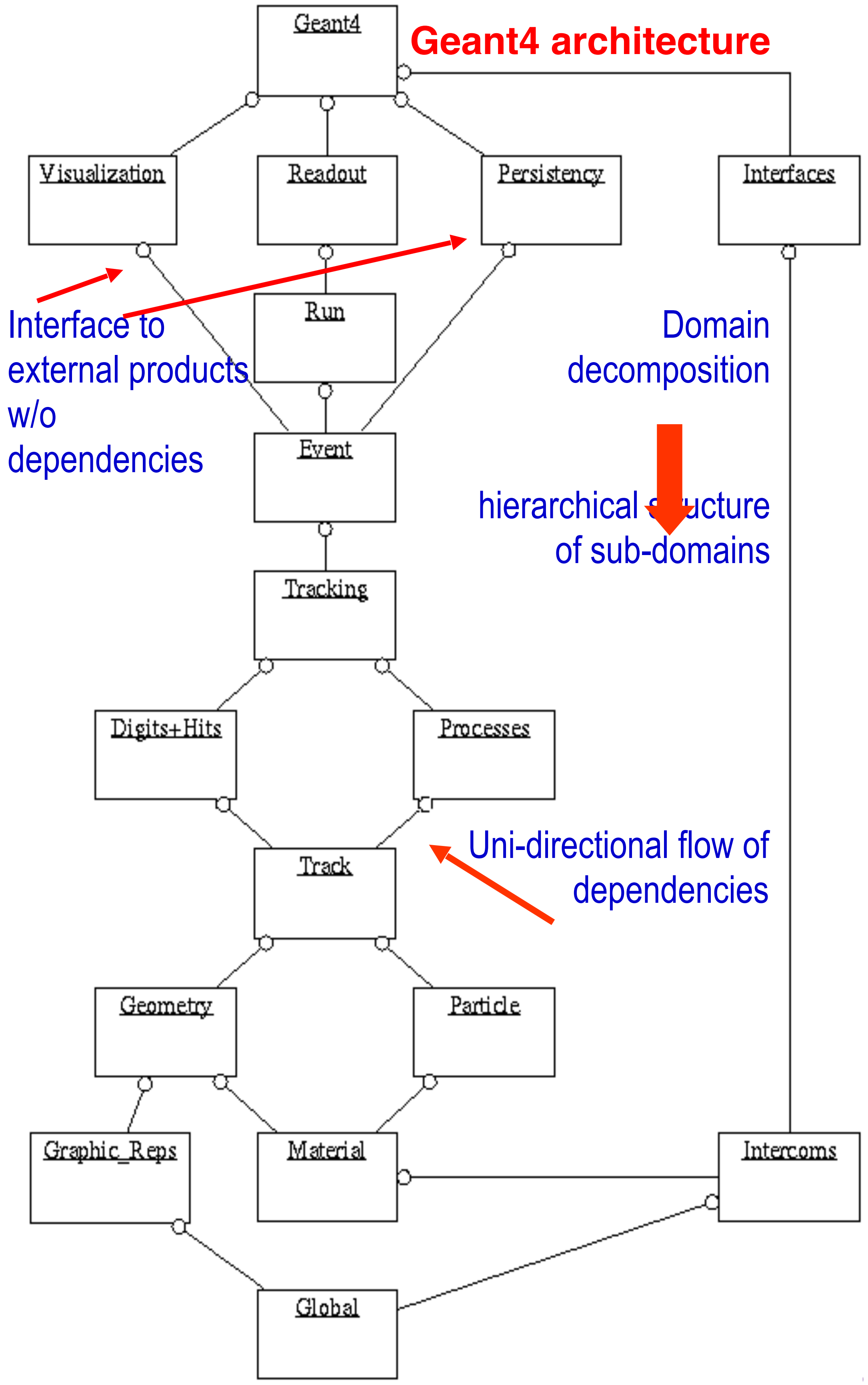
- Geant4 provides tools for particle transport and also to model experimental environments

- The "User" needs to use Geant4 tools
  - to tell the Geant4 kernel about the simulation configuration
  - to interact with the Geant4 kernel itself

- The "User" must tell Geant4 what he/she only knows
  - The experimental scenario
    - Geometry, materials, sensitive and passive elements
    - Primary particles, radiation environment
  - That the "User" wants to happen during transport
    - which particles are to be tracked
    - which physics processes would be of interest (*and which options for modelling are preferable*)
    - how precise the simulation is going to be

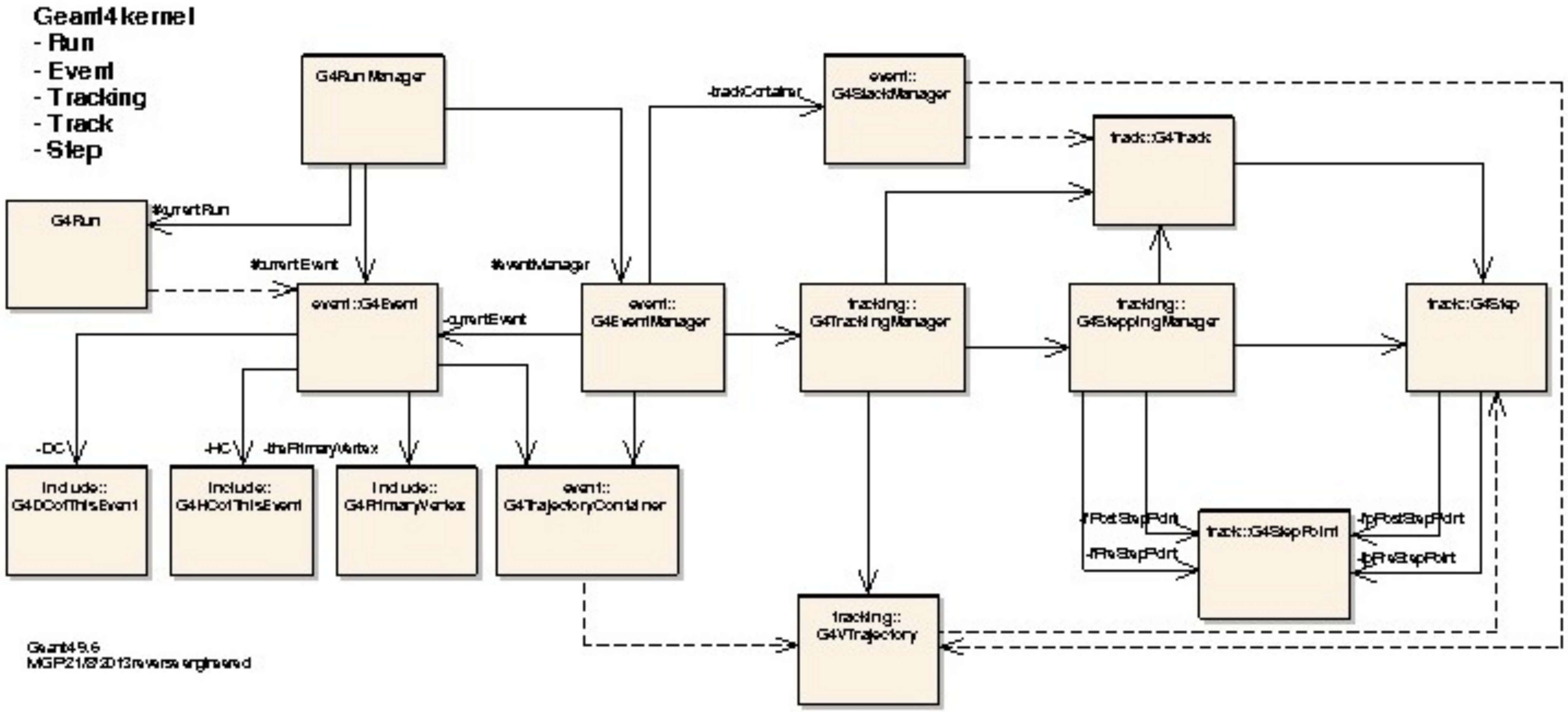
Code Arrangement

- All codes of Geant4 are grouped into 17 categories
- Relationships among classes from two categories:
  - Always one sided
  - No cyclic dependencies
- "Global" is at the lowest level
  - No dependency on any class from other categories of Geant4
- "Geant4" is at the highest level
  - Classes in this category depend on all other categories
- External dependencies:
  - CLHEP (HEP utilities)
  - PTL (for making libs/executable)
  - EXPAT | XML
  - XERCES | parsing
  - … some Graphics packages …

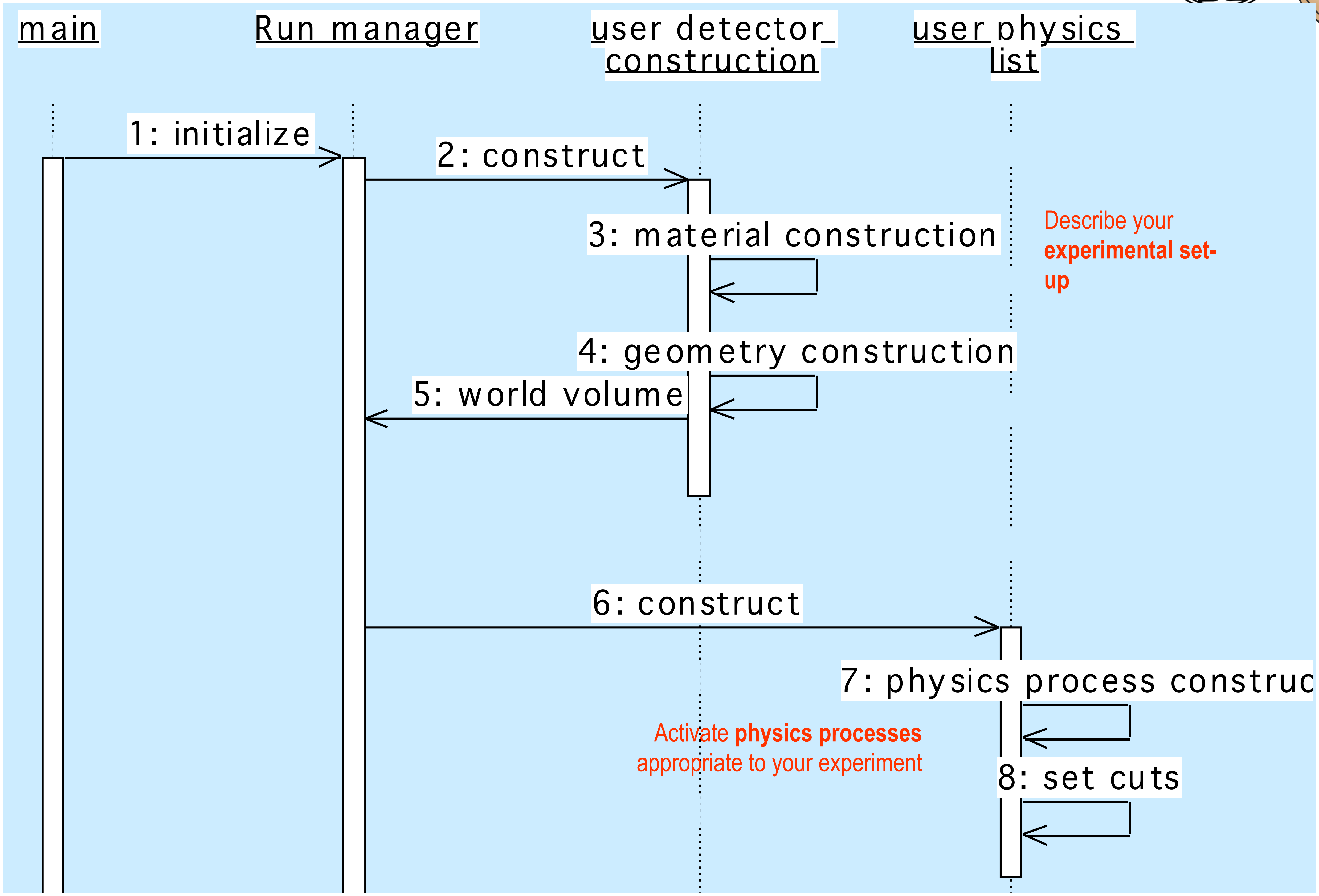


Some of the terminologies

- Run, Event, Track, Step, Stack, …
- Track vs. Trajectory;   Step vs. Trajectory Point
- Particle, Process, Hits, …



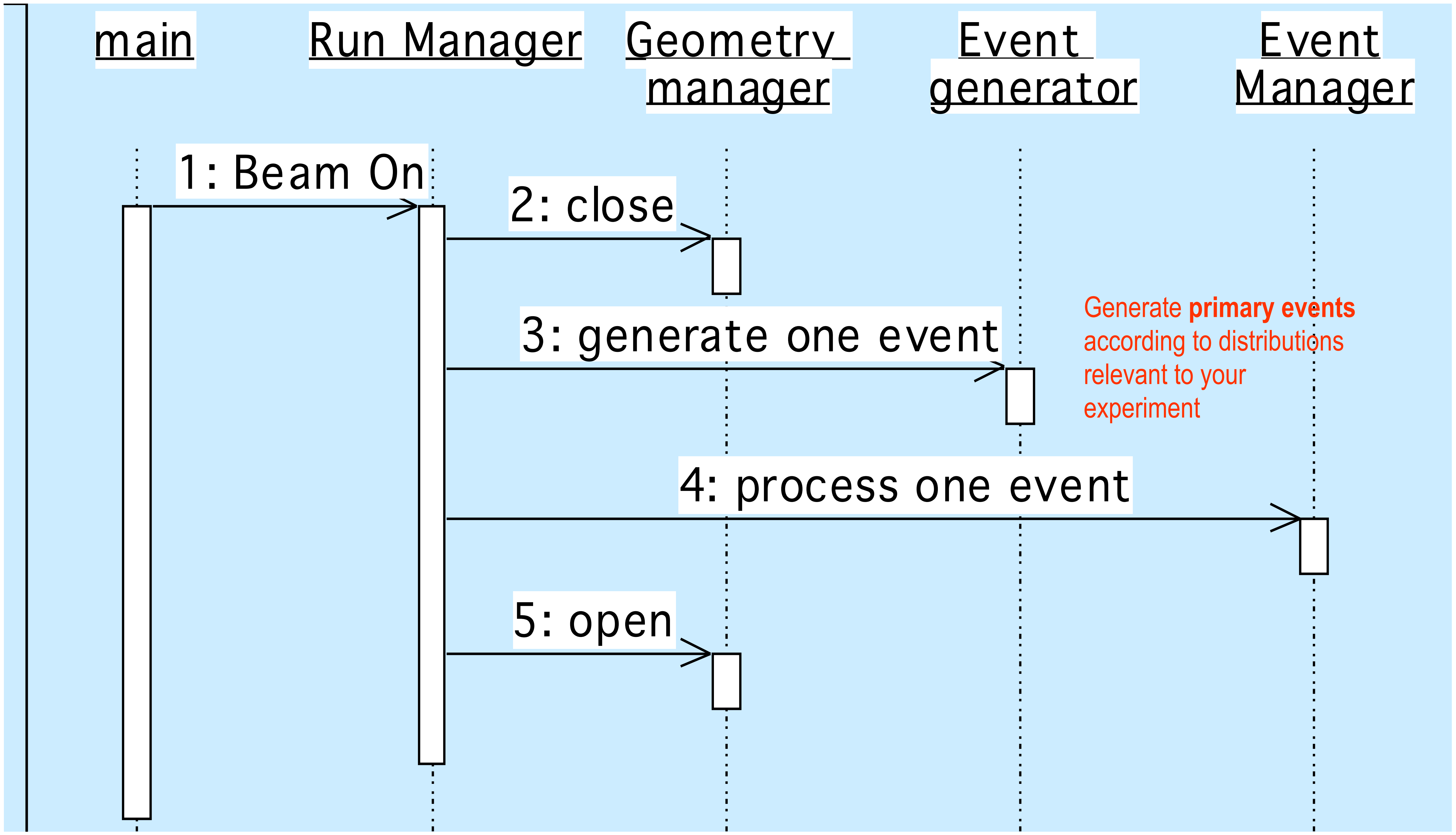
Initialization



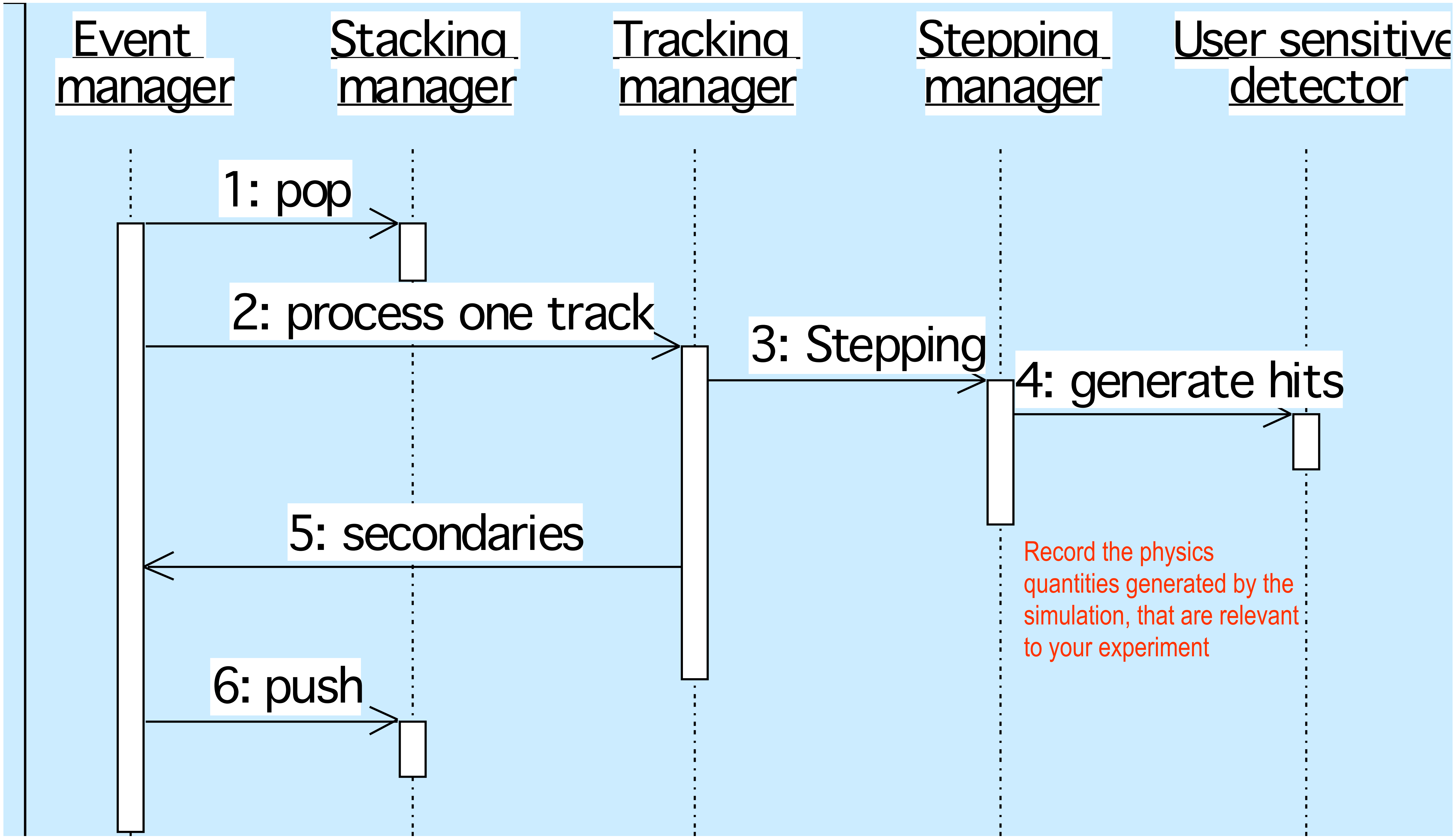
Describe your **experimental set-up**

Activate **physics processes** appropriate to your experiment

Event Loop


main
Run Manager
Geometry manager
Event generator
Event Manager
1: Beam On
2: close
3: generate one event
4: process one event
5: open
Generate **primary events** according to distributions relevant to your experiment

Event Processing



Record the physics quantities generated by the simulation, that are relevant to your experiment

Run in Geant4

- Conceptually, a Run is a collection of Events which share the same detector setup and physics conditions
  - A Run consists of one loop over Events

- Within a Run, the user cannot change
  - the detector setup
  - settings of the physics processes

- As an analogy of the real experiment, a run of Geant4 starts with “Beam On”

- At the beginning of a Run, the geometry is optimised for navigation and cross-section tables are calculated according to materials which appear in the geometry and with the defined cut-off values

- G4RunManager class manages the processing of a Run, a Run is represented by the G4Run class or a user-defined class derived from it

- G4UserRunAction is an optional user hook

Event in Geant4

- An Event is the basic unit of simulation in Geant4
- At the beginning of processing, primary tracks are generated. These tracks are pushed into a stack
- A track is popped up from the stack one by one and is traced through the detector. The resulting secondary tracks, if any, are pushed into the stack
  - This “tracking” lasts as long as the stack has a track
- When the stack becomes empty, the processing of the event is over
- G4Event class represents an Event. It has the following objects at the end of its (successful) processing
  - List of primary vertices and particles (as input)
  - Hits and trajectory collections (as output)
- G4EventManager class manages the processing of an event
- There is an optional user hook: G4UserEventAction

Track in Geant4

- A Track is a snapshot of a particle
  - It has physical quantities of the current instance only. It does not contain a record of previous quantities
  - A Step is a “delta” information of a Track. A Track is not a collection of Steps. Instead, a Track is being updated by the Step.

- The Track object is deleted when
  - it goes out of the world volume,
  - it disappears (through decays, inelastic scattering, …),
  - it goes down to zero kinetic energy and no “AtRest” additional process is required for the particle, or
  - the user decides to kill it artificially.

- No Track object persists at the end of an event
  - For the record of Tracks, use Trajectory class objects

- G4TrackingManager manages the processing of a Track. A Track is represented by the G4Track class

- There is an optional user hook: G4UserTrackingAction

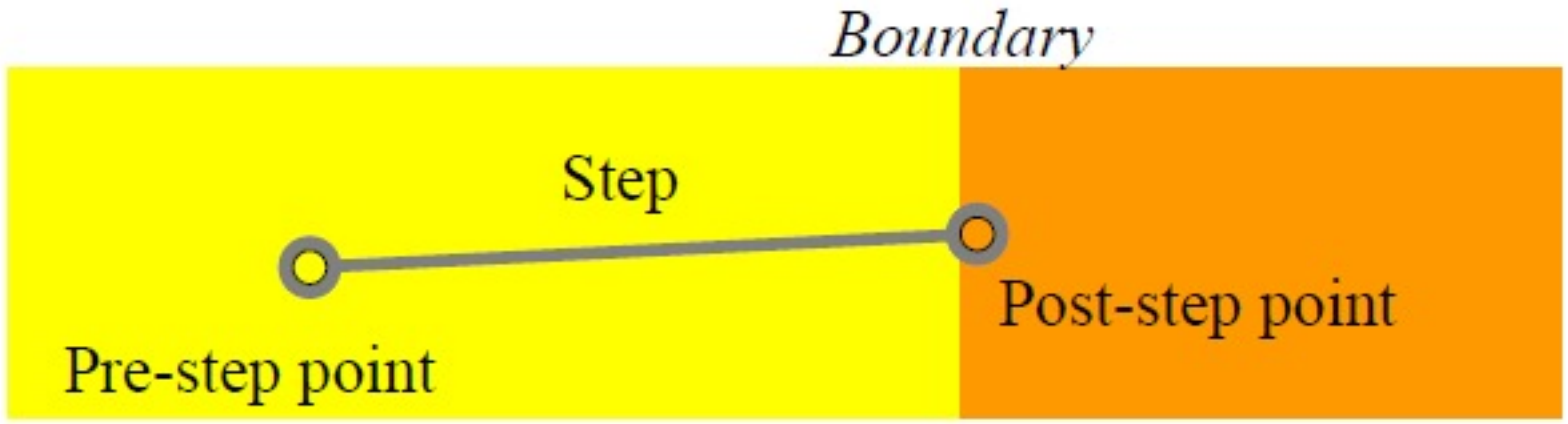
Step in Geant4

- A Step has two points and also "delta" information of a particle (energy loss in the step, time-of-flight spent by the step, etc.)

- Each point knows the volume (and it's material) where it is in. In case a step is limited by a volume boundary, the endpoint will physically stand on the boundary, and it logically belongs to the next volume
  - Since each Step knows materials of two volumes, boundary processes such as transition radiation or reflection could be simulated

- G4SteppingManager class manages the processing of a Step, and a Step is represented by the G4Step class

- G4UserSteppingAction is the optional user hook



Trajectory and Trajectory Point

- Please remember, a Track does not keep its trace and no Track object persists at the end of an Event

- G4Trajectory is the class which copies some of the information of a G4Track object. Likewise, G4TrajectoryPoint is the class which keeps some of the information of a G4Step object
  - A G4Trajectory object has a vector of G4TrajectoryPoint objects
  - At the end of event processing, the G4Event object has a collection of G4Trajectory objects provided
    - /tracking/storeTrajectory is set to 1

- Keep in mind the distinction:
  - G4Track vs G4Trajectory,   G4Step vs G4TrajectoryPoint

- Given that the G4Trajectory and G4TrajectoriyPoint objects persist till the end of an event, one should be careful not to store too many trajectories
  - Avoid storing shower tracks from a high-energy particle

- G4Trajectory and G4TrajectoryPoint objects store only the minimum information
  - The user can create his/her own Trajectory/TrajectoryPoint classes to store the required information. These classes can be derived from the base classes G4VTrajectory and G4VTrajectoryPoint

Particle in Geant4

- A particle in Geant4 is represented by three layers of classes:
  - G4Track:
    - Position, geometrical information, etc.
    - This is a class representing a particle to be tracked
  - G4DynamicParticle:
    - “Dynamic” physical properties of a particle, such as momentum, energy, spin, etc.
    - Each G4Track object has its own unique G4DynamicParticle Object
    - This is a class representing an individual particle
  - G4ParticleDefinition:
    - “Static” properties of a particle, such as charge, mass, lifetime, decay channels, etc.
    - G4ProcessManager which describes the processes involving the particles
    - All G4DynamicParticle objects of the same kind of particles share the same G4ParticleDefinition

Tracking and Process

- Tracking in Geant4 is universal
  - It is independent of
    - the particle type
    - the physics processes involving the particle
  - It gives the chance to all processes
    - to contribute to the determination of the step length
    - to contribute any possible changes in physical quantities of the track
    - to generate secondary particles
    - to suggest changes in the state of the track
      - e.g. to suspend, postpone or kill it

Process in Geant4

- In Geant4, particle transportation is a process as well, by which a particle interacts with geometrical volume boundaries and fields of any kind
  - Because of this, the shower parametrization process can take over from ordinary transportation without modifying the transportation process

- Each particle has its own list of applicable processes. At each step, all processes involved are invoked to get proposed physical interaction lengths

- The process which requires the shortest interaction length (in space-time) limits the step

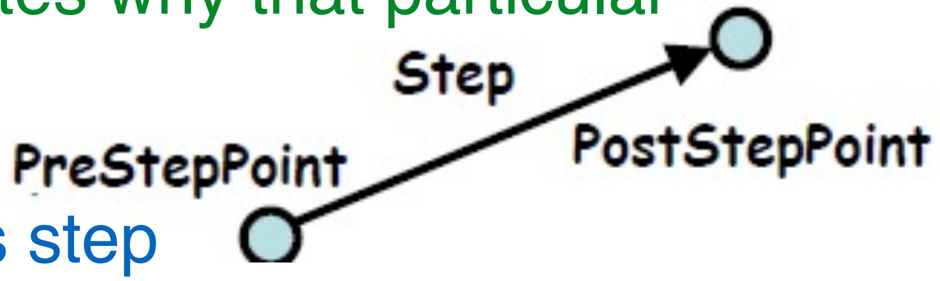
- Each process has one or combination of actions with the following nature:
  - At Rest
    - e.g. muons can decay at rest
  - Along Step (a.k.a. continuous process)
    - e.g. Cherenkov process
  - Post step (a.k.a. discrete process)
    - e.g. decay in flight

Track Status

- At the end of each step, the state of a track may change (according to the processes involved)
  - The user can also change the status in UserSteppingAction
  - Status, as mentioned below, are artificial, i.e. Geant4 kernel won't set them, but the user can
    - fAlive
      - continue the tracking
    - fStopButAlive
      - the track has come to zero kinetic energy, but still AtRest process to occur
    - fStopAbdKill
      - The track has lost its identity because it has decayed, interacted, or gone beyond the world boundary
      - Secondaries will be pushed to the stack
    - fKillAndSecondaries
      - kill the current track and also associated secondaries
    - fSuspend
      - suspend the processing of the current track and push it and its secondaries to the stack
    - fPostponeToNextEvent
      - Postpone processing of the current track to the next events
      - Secondaries are still being processed within the current event

Step Status

- The step status attached to the G4StepPoint indicates why that particular step was chosen
  - "PostStepPoint" gives the status of this step
  - "PreStepPoint" provides the status of the previous step
    - fWorldBoundary
      - step reached the world boundary
    - fGeomBoundary
      - step is limited by a volume boundary except for the world
    - fAtRestDoItProc, fAlongStepDoItProc, fPostStepDoItProc
      - step is limited by AtRest, AlonStep or PostStep process
    - fUserDefineLimit
      - step is limited by the user step limit
    - fExclusiveForcedProc
      - step is limited by an exclusively forced process (e.g. shower parametrisation)
    - fUndefined
      - step not defined
- If the first step in a volume is to be identified, pick fGeomBoundary status in the PreStepPoint
- If a step going out of a volume is to be identified, pick fGeomBounday status in the PostStepPoint

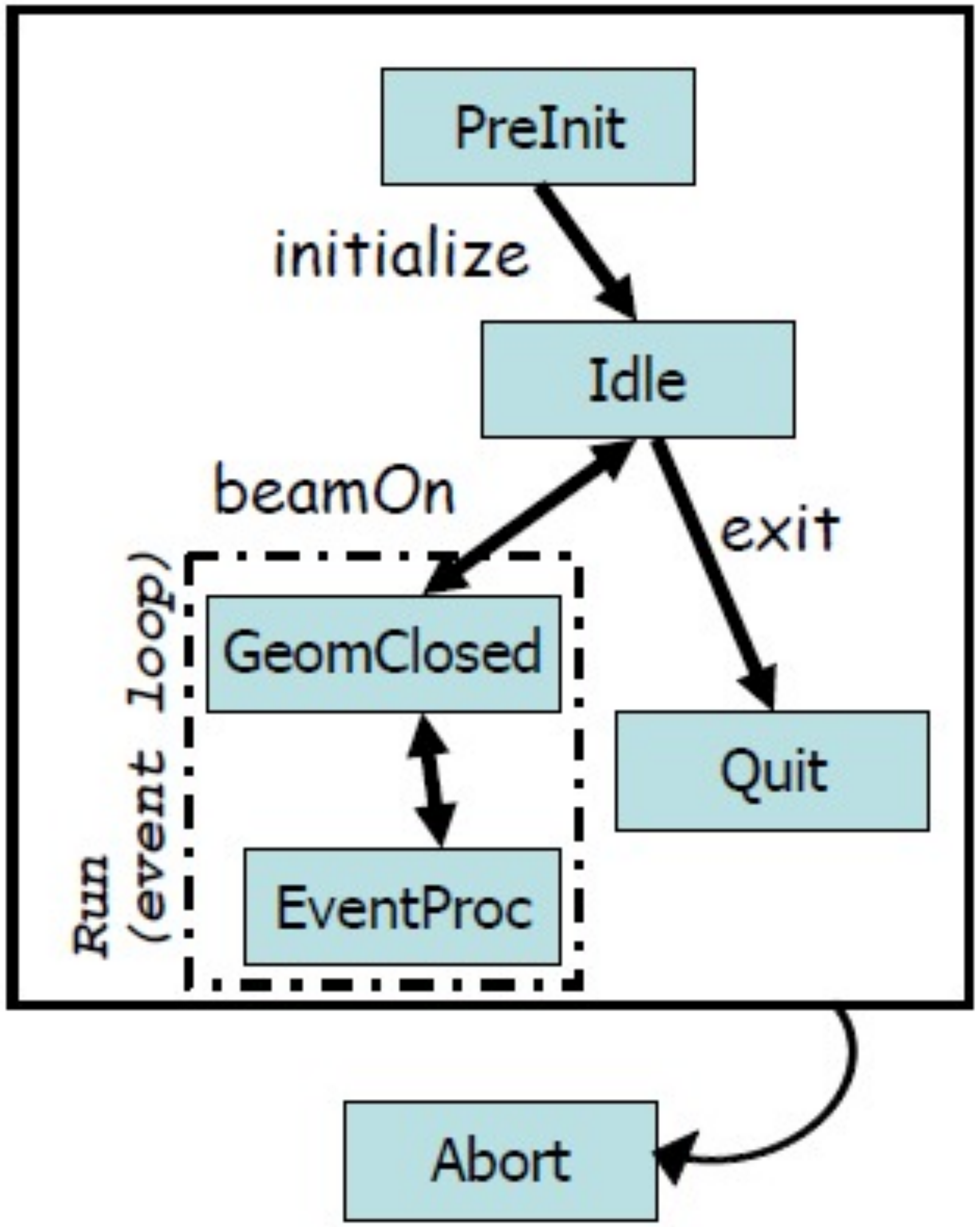


Extraction of Useful Information

- Given geometry, physics and primary track information, Geant4 does proper physics simulation “silently”
  - The user has to add a bit of code to extract useful information
- There are two ways for extraction:
  - Use the user hooks provided by Geant4
    - These are: G4UserTrackingAction, G4UserSteppingAction, ….
      - The user has access to almost all information
      - It is straight-forward but do-it-yourself
  - Use Geant4 scoring functionality
    - Assign G4VSensitiveDetector to a volume
    - Hits collection is automatically stored in the G4Event object, and automatically accumulated if the user-defined Run object is used
    - Use user hooks to get event/run summary
      - The relevant action classes are G4UserEventAction, G4UserRunAction

Geant4 as a State Machine

- Geant4 has six application states:
  - G4State_PreInit
    - At this state, material, geometry, particle and physics process need to be defined and initialized
  - G4State_Idle
    - Geant4 is ready to start a run
  - G4State_GeomClosed
    - Geometry is optimised and ready to process an event
  - G4State_EventProc
    - An event is being processed
  - G4State_Quit
    - (Normal) termination
  - G4State_Abort
    - A fatal exception occurred and the program is aborting



Track Stacks in Geant4

- By default, Geant4 has three track stacks:
  - "Urgent", "Waiting" and "PostponeToNextEvent"
  - Each stack operates in a simple "last-in-first-out" mode
  - User can increase arbitrarily the number of stacks

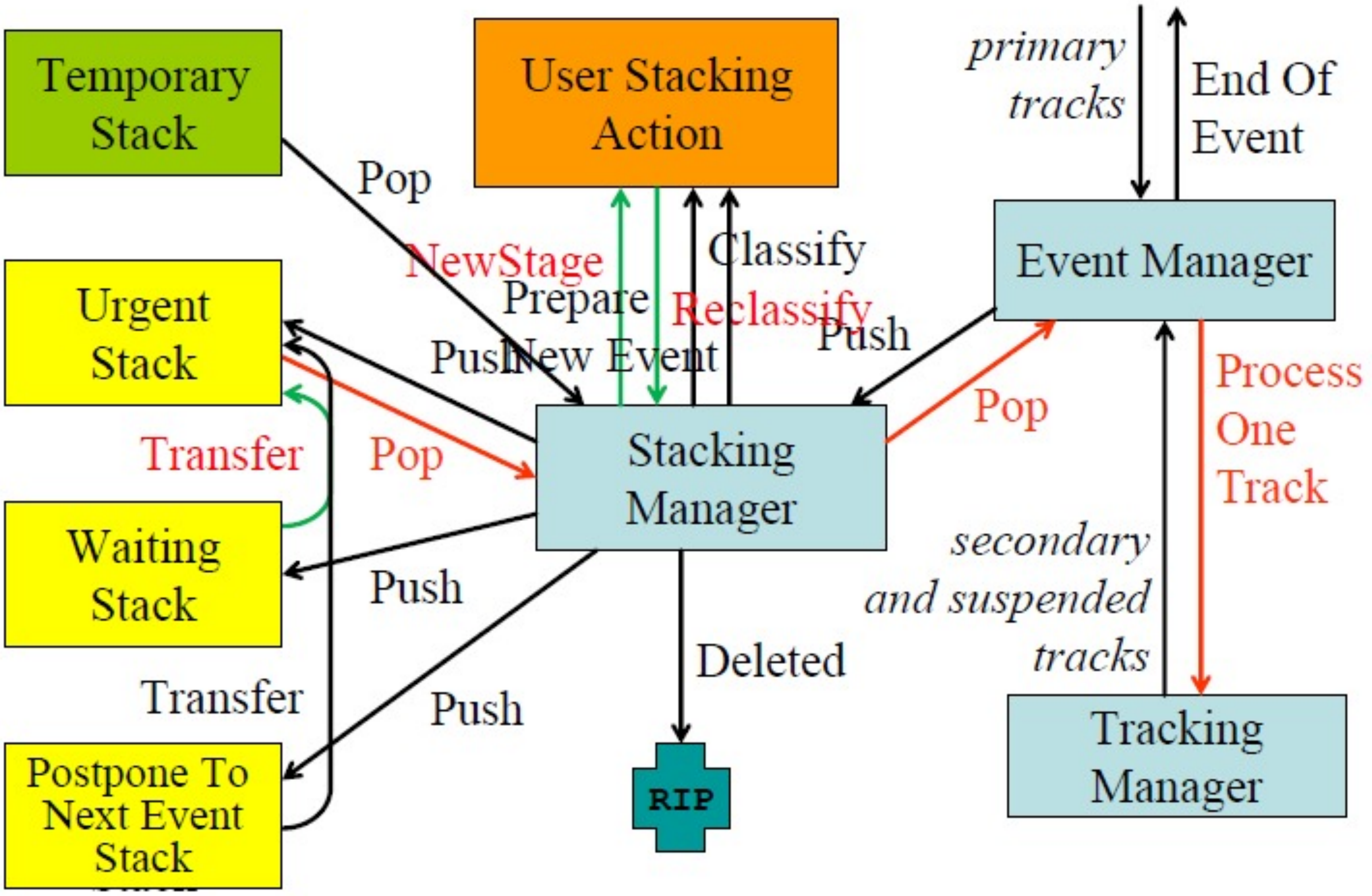
- ClassiftNewTrack() method of G4UserStackingAction class decides which stack each newly created secondary particle to be stored (or be killed)
  - By default, all tracks go to the "Urgent" stack

- A G4Track is popped up only from the "Urgent" stack

- Once the "Urgent" stack is empty, all tracks in the "Waiting" stack are transferred to the "Urgent" stack
  - And NewStage() method of th4 G4UserStackingAction is invoked

- Utilising more the one stack, the user can control the priorities of processing tracks without paying the overhead of "scanning the highest priority track"
  - Proper selection/abortion of tracks/events with well-designed stack management provides significant efficiency increase of the entire simulation

Stacking Mechanism



Tips of Stacking Manipulation

- Classify all secondaries as fWaiting until Reclassify() method is invoked
  - One can simulate all primaries before any secondary

- Classify secondary tracks below a certain energy as fWaiting until the Reclassify() method is invoked
  - One can roughly simulate the event before being bothered by low energy electromagnetic showers

- Suspend a track on its fly. Then this track and all of the already generated secondaries are pushed to the stack
  - Given the stack is “last-in-first-out”, secondaries are popped out prior to the original suspended track
  - This is quite effective for simulating Cerenkov radiation

- Suspend all tracks that are leaving a region, and classify these suspended tracks as fWaiting until Reclassify() method is invoked
  - One can simulate all tracks in this region prior to other regions
  - Note that some backsplash tracks may come back into this region later

Primary Generator

- Each Geant4 Event starts with the generation of one or multiple primary particles

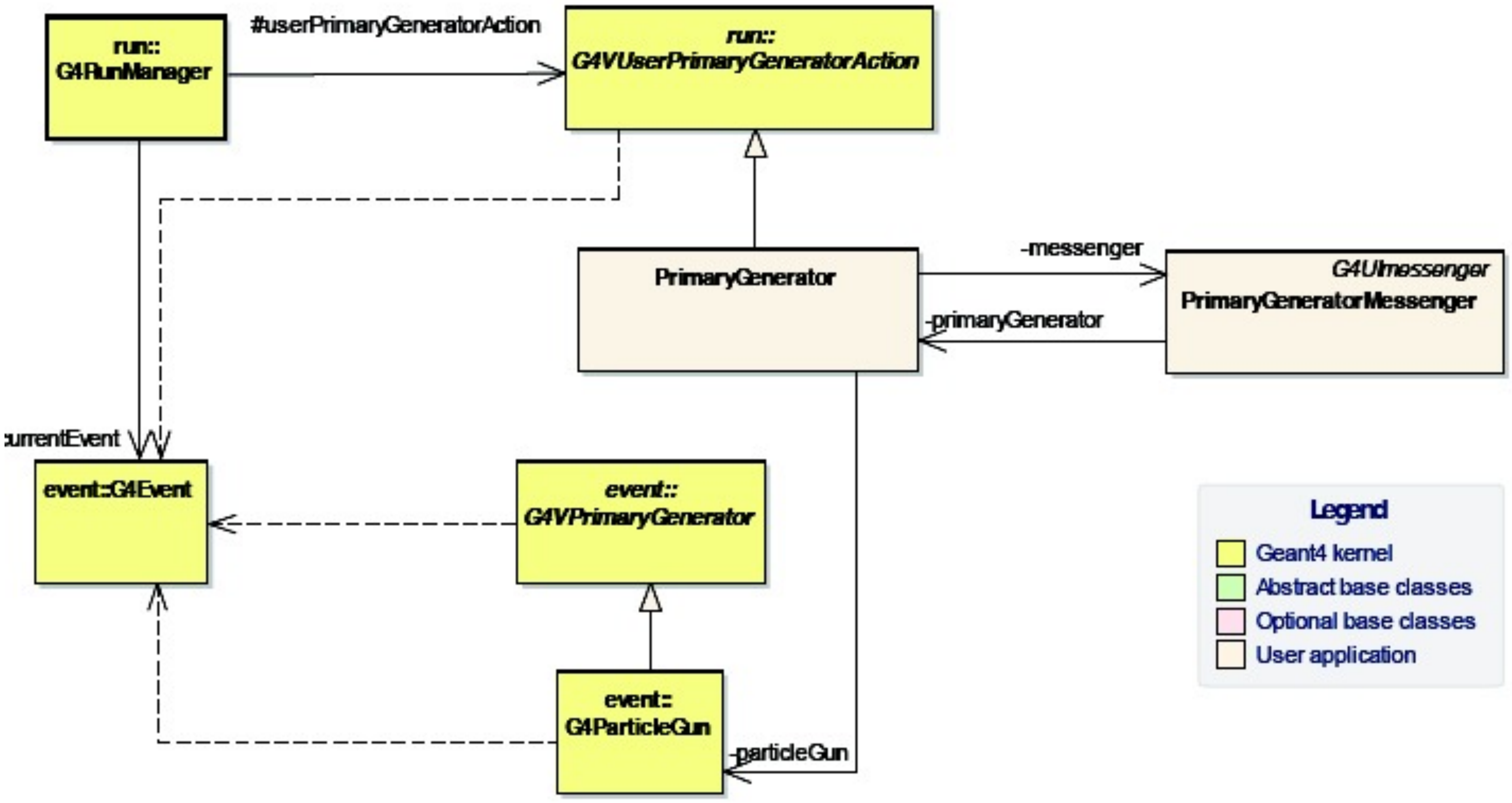
- The user has to define the properties of primary particles
  - Particle type, e.g. electron, gamma, ion, …
  - Initial kinematics, e.g. energy, momentum, origin, …
  - Additional properties, e.g. polarization, …

- These properties can be divided into:
  - G4PrimaryVertex: specifying start point in space and time
  - G4PrimaryParticle: specifying initial momentum, polarisation, PDG code, list of daughters for decay chains

- A primary generator is a class derived from G4VPrimaryGenerator and has an implementation of the method GeneratePrimaryVertex()
  - The primary vertex and the primary particle(s) are added in this method to a Geant4 Event
  - Several event generators are provided in the Geant4 toolkit
    - G4HEPEvtInterface, G4HEPMCInterface, G4GeneralParticleSoce, G4ParticleGun

PrimaryGeneratorAction

- This mandatory user action controls the generation of primary particles but does not generate the primaries itself. This task is delegated to G4PrimaryGenerator derived from G4VPrimaryGenerator



PrimaryGenerator

- The primary generator could make use of
  - a built-in generator: G4ParticleGun for producing particles of arbitrary momenta and direction to test a setup
  - an external physics generator which provides a collection of particles emerging from a known physics process (Pythia, Herwig, …)

```
PrimaryGeneratorAction::PrimaryGeneratorAction(const G4String & parName,
G4double energy, G4ThreeVector pos, G4ThreeVector momDirection){

        const G4int nParticles = 1;
        fParticleGun = new G4ParticleGun(nParticles);
        G4ParticleTable* parTable = G4ParticleTable::GetParticleTable();
        G4ParticleDefinition* parDefinition = parTable->FindParticle(parName);
        fParticleGun->SetParticleDefinition(parDefinition);
        fParticleGun->SetParticleEnergy(energy);
        fParticleGun->SetParticlePosition(pos);
        fParticleGun->SetParticleMomentumDirection(momDirection);
}
```

```
PrimaryGeneratorAction::GeneratePrimaries(G4Event* evt){

        //some additional random sampling here

        fParticleGun->GeneratePrimaryVertex(evt);

}
```

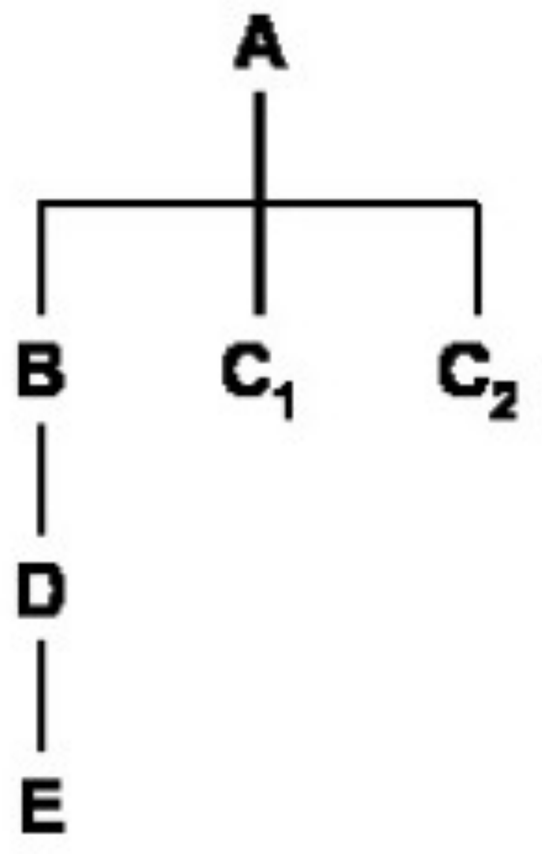
Attaching user information

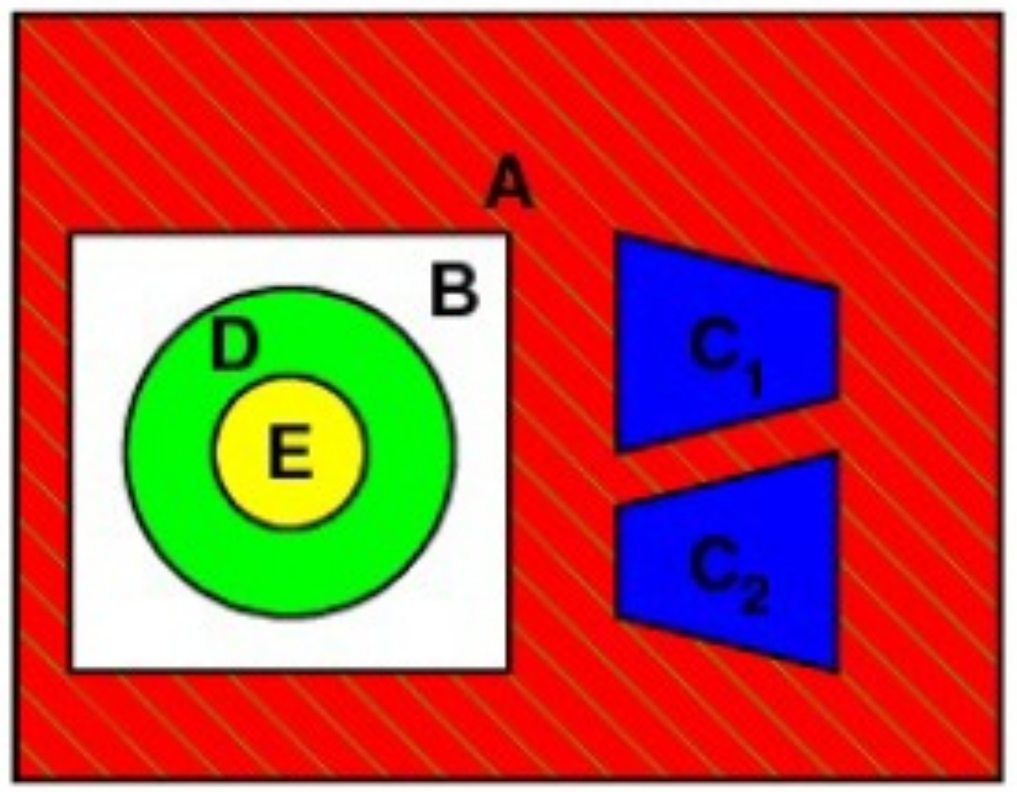
- Abstract classes:
  - The user can use his/her own class derived from the period base class
  - G4Run, G4VHit, G4VDigit, G4VTrajectory, G4VTrajectoryPoint
- Concrete classes:
  - The user can attach a user information class object
    - G4Event — G4VUserEventInformation
    - G4Track — G4VUserTrackInformation
    - G4PrimaryVertex — G4VUserPrimaryVertexInformation
    - G4PrimaryParticle — G4VUserPrimaryParticleInformation
    - G4Region — G4VUserRegionInformation
  - User information class objects are deleted when the associated Geant4 class object is deleted

Modelling of a Detector

- Detectors are modelled by a geometrical shape filled with certain material of known physical properties → “Volume”

- Several volumes can describe different components of the detector system. Put them together in a hierarchical structure:





Composite Volume ⟶ Experimental Setup

Material

- Material has a Name, Effective Atomic Number and Weight, Density, Radiation (L_R) and Absorption (λ) length
- It can be defined by specifying these attributes
- If radiation and absorption lengths are not known, one can furnish information regarding the chemical composition and Geant4 will compute the required attributes for the application
- One can also add the state (solid I liquid I gas), isotopic properties, etc. Some of these specifications are essential to study activation, …
  - Define pseudo-elements:
    new G4Material (name, Z, A, density, state, temperature, pressure);
  - Define a mixture of elements in atomic or weighted proportion:
    new G4Material (name, density, numComponents);
      →AddElement (material, fraction);
      G4Element* element = new G4Element (name, symbol, Z,   A);
      →AddElement (element, nAtom);
  - Build elements from isotopes:
    new G4Element (name, symbol, numIsotopes);
      G4Isotope* isotope = new G4Isotope (name, Z, N);
      →AddIsotope (isotope, abundance);

Volume

- A volume is defined by its shape, dimensional parameters and its material content. A shape with dimensional parameters is called a Solid and the association of a Solid with a Material is called a LogicalVolume.

- There are several ways to define a Solid:
  - Computed Solid Geometry (CSG): G4Box, G4Trd, G4Trapm G4Tubs, G4Cons, G4Sphere, G4PolyCone, ……..
  - Boundary Representation (BREP): G4BrepSolidPcone, …….. (slower navigation on a standard CPU, but works better in GPU)
  - Boolean: Solids made out by adding, subtracting, intersecting, etc. several solids: G4RotateSolid, ……
  - STEP: Imported from the CAD system

- System of Units: Though a convention is used internally for the unit system, the recommendation is not to remember them and use the units explicitly:

  double length = 5 * cm;
  double angle = 30 * deg;
  double time = 25 * ns;

Define a Volume

- First define a Material, say Air consisting of 2 constituent elements Nitrogen and Oxygen in a given weight proportion (70:30)

  - First define Nitrogen and Oxygen with their appropriate Z, A values:

    ```
    G4Element *eln = new G4Element(“Nitrogen”,”N”,Z=7,A=14.01*g/mole);
    G4Element *elo = new G4Element(“Oxygen”,”O”,Z=8,A=16.00*g/mole);
    ```

  - Then define Air and add the two elements:

    ```
    G4Material *air = new G4Material(“Air”,1.205E-03*g/cm3,2);
    air->AddElement(eln,0.7);
    air->AddElement(elo,0.3);
    ```

- Define a Solid of a given name (say INOM) as a box of given half length, half width, half thickness:

  ```
  G4Solid* inomSolid = new G4Box(“INOM”,1606*cm,706*cm,598*cm);
  ```

- Associate the Solid with the Material to define the LogicalVolume:

  ```
  G4LogicalVolume* inomLog = new G4LogicalVolume(inomSolid,air,”INOML”);
  ```

- The reference frame is a right-handed Cartesian coordinate system with the origin at the centre of the box

Define a Detector Setup

- To define a set up, one needs to
  - define a Master or World reference system
  - position the various components with respect to each other

- Geant4 uses the concept of PhysicalVolume which is a LogicalVolume positioned in a Mother (PhysicalVolume or LogicalVolume) with a translation vector and a rotation matrix (optional). For the top-level Volume (defining the World reference system), the pointer of the Mother Volume should be declared as a nullPointer

- One useful way of defining daughter volumes is by dividing an existing mother volume into n equal parts along a chosen axis (Cartesian, Cylindrical, or Polar)
  - The creation and positioning are done in two separate steps

- When a daughter is positioned inside a mother, the extent inside the mother occupied by the daughter gets filled with the material of the daughter's volume

Can build up a tree like a Russian doll

Hooks for Positioning

```
G4PhysicalVolume* volume = new G4PVPlacemenet (rot,
                    G4ThreeVector(xpos*cm,ypos*cm,zpos*cm),
                    G4LogicalVolume* current,  Name,
                    G4LogicalVolume* mother,  false, copyNumber)
```

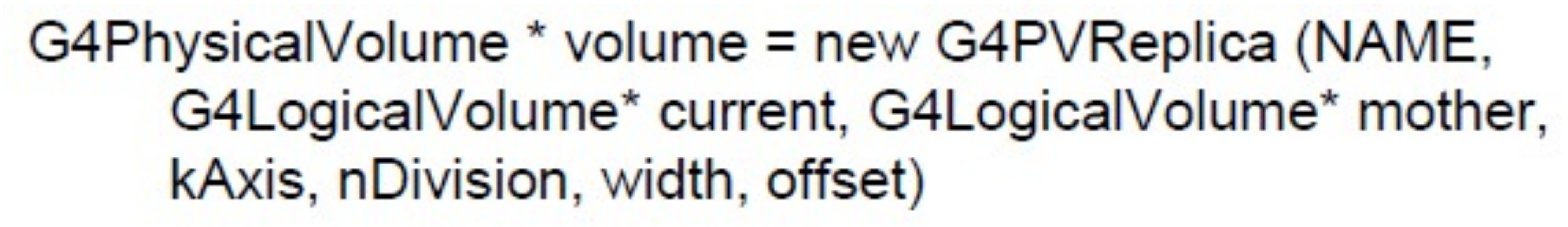
creates a PhysicalVolume volume by positioning a copy with number copyNumber of the LogicalVolume current inside the mother volume mother with a translation vector (G4ThreeVector) and a rotation matrix rot (G4RotationMatrix*)

If one needs to define a rotation matrix by specifying the angles (θ_i,φ_i) of the three axes (as done in Geant3), one needs to follow the steps:

```
G4ThreeVector iAxis(sin(thetaI*deg)*cos(phiI*deg),
                            sin(thetaI*deg)*sin(phiI*deg), cos(thetaI*deg));
G4RotationMatrix* rot = new G4RotationMatrix();
rot->rotateAxis(xAxis,yAxis,zAxis);
rot->invert();
```

Hooks for Positioning (2)

- For dividing a parent volume into equal parts along a given axis, one needs to create the LogicalVolume using the standard steps (defining Solid, Material, and LogicalVolume) and then position multiple replicas through:



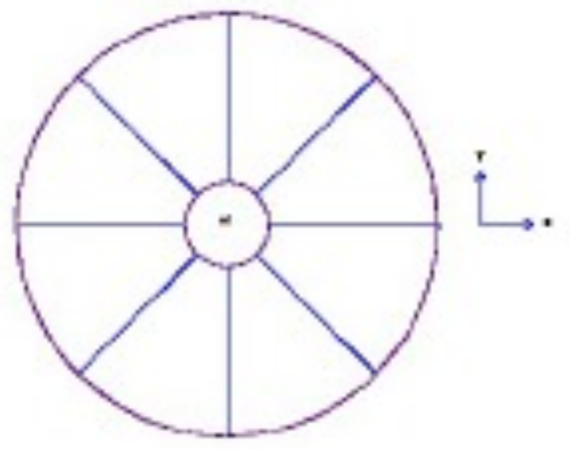
```
G4PhysicalVolume * volume = new G4PVReplica (NAME,
        G4LogicalVolume* current, G4LogicalVolume* mother,
        kAxis, nDivision, width, offset)
```

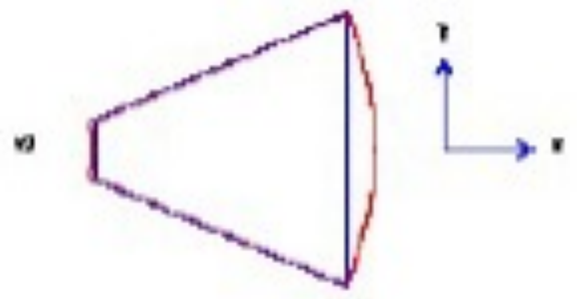
  This needs more steps than what used to be in its predecessor Geant3 (GSDVN), but it is more general

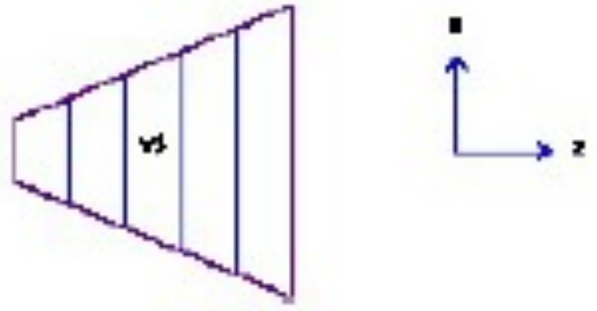
- The tree of PhysicalVolumes is instantiated at the time of tracking in the form of G4VTouchable and the list of PhysicalVolumes in the branch will provide the unique identification of a volume

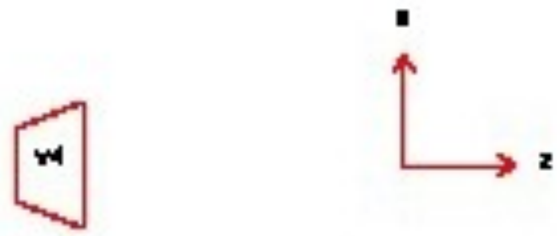
Example

- Construct geometry of a cylindrical drift chamber with 8 sectors each having 5 cells:









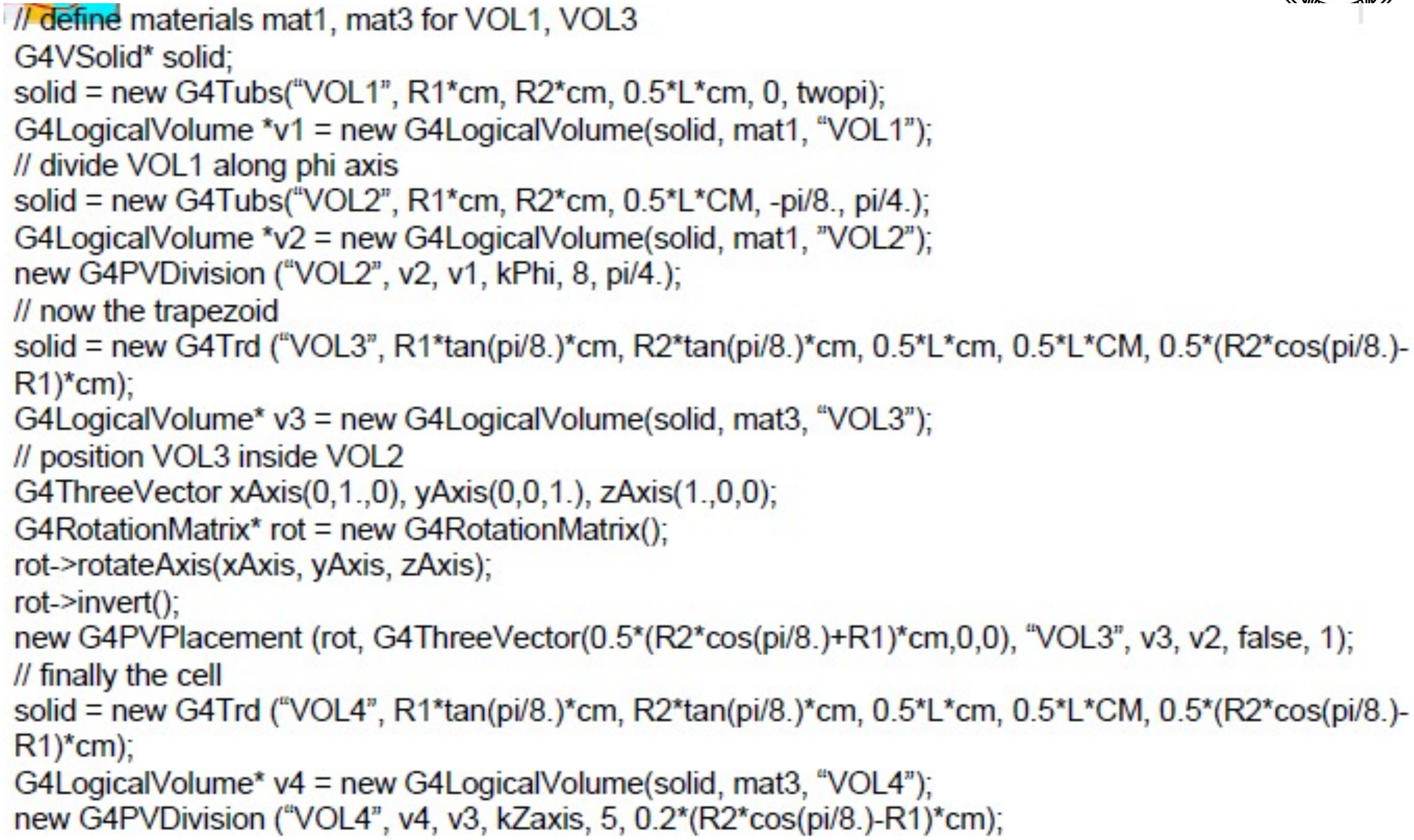
- Define volume VOL1 as a tube with inner and outer radii R1, R2 and half-length L

- Divide the tube into 8 parts azimuthally and each section is called VOL2

- Define a trapezoid of half-length L, width [R2.cos(π/8)—R1] and two edges of dimension 2.R1.tan(π/8), and 2.R2.tan(π/8). Position this volume VOL3 inside VOL2 with the proper translation vector and rotation matrix

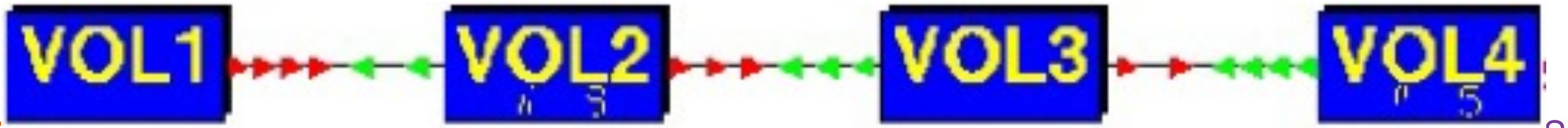
- Divide the trapezoid VOL3 into 5 parts along the z-axis. Each part VOL4 will be a cell

Example

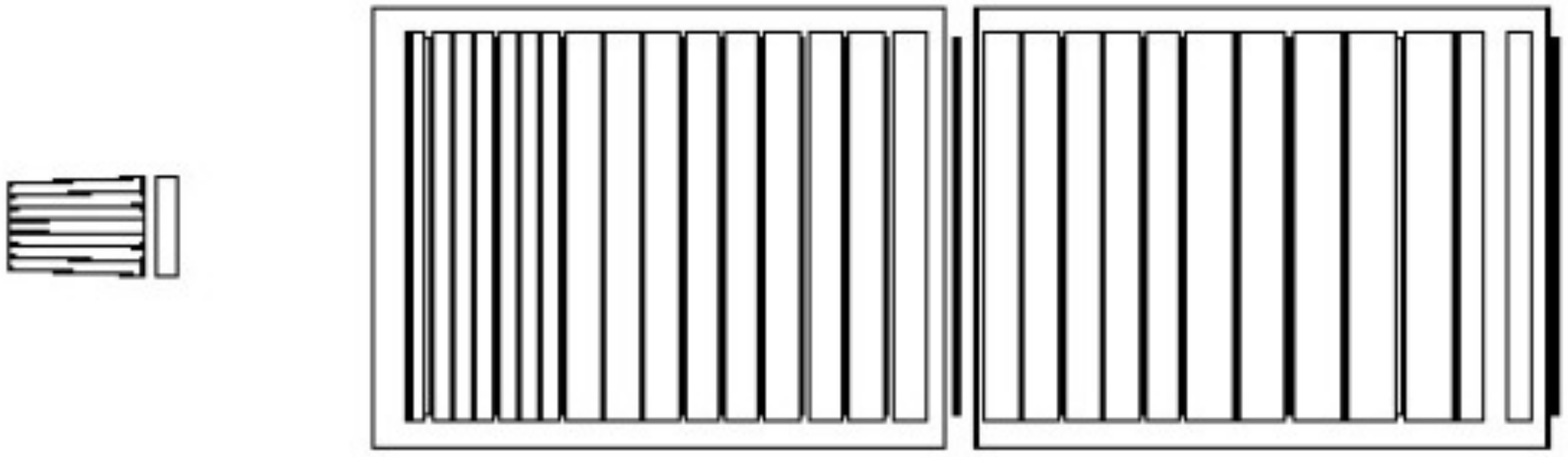


```
// define materials mat1, mat3 for VOL1, VOL3
G4VSolid* solid;
solid = new G4Tubs(“VOL1”, R1*cm, R2*cm, 0.5*L*cm, 0, twopi);
G4LogicalVolume *v1 = new G4LogicalVolume(solid, mat1, “VOL1”);
// divide VOL1 along phi axis
solid = new G4Tubs(“VOL2”, R1*cm, R2*cm, 0.5*L*CM, -pi/8., pi/4.);
G4LogicalVolume *v2 = new G4LogicalVolume(solid, mat1, ”VOL2”);
new G4PVDivision (“VOL2”, v2, v1, kPhi, 8, pi/4.);
// now the trapezoid
solid = new G4Trd (“VOL3”, R1*tan(pi/8.)*cm, R2*tan(pi/8.)*cm, 0.5*L*cm, 0.5*L*CM, 0.5*(R2*cos(pi/8.)-
R1)*cm);
G4LogicalVolume* v3 = new G4LogicalVolume(solid, mat3, “VOL3”);
// position VOL3 inside VOL2
G4ThreeVector xAxis(0,1.,0), yAxis(0,0,1.), zAxis(1.,0,0);
G4RotationMatrix* rot = new G4RotationMatrix();
rot->rotateAxis(xAxis, yAxis, zAxis);
rot->invert();
new G4PVPlacement (rot, G4ThreeVector(0.5*(R2*cos(pi/8.)+R1)*cm,0,0), “VOL3”, v3, v2, false, 1);
// finally the cell
solid = new G4Trd (“VOL4”, R1*tan(pi/8.)*cm, R2*tan(pi/8.)*cm, 0.5*L*cm, 0.5*L*CM, 0.5*(R2*cos(pi/8.)-
R1)*cm);
G4LogicalVolume* v4 = new G4LogicalVolume(solid, mat3, “VOL4”);
new G4PVDivision (“VOL4”, v4, v3, kZaxis, 5, 0.2*(R2*cos(pi/8.)-R1)*cm);
```

- This will create the volume tree:



A Practical Example

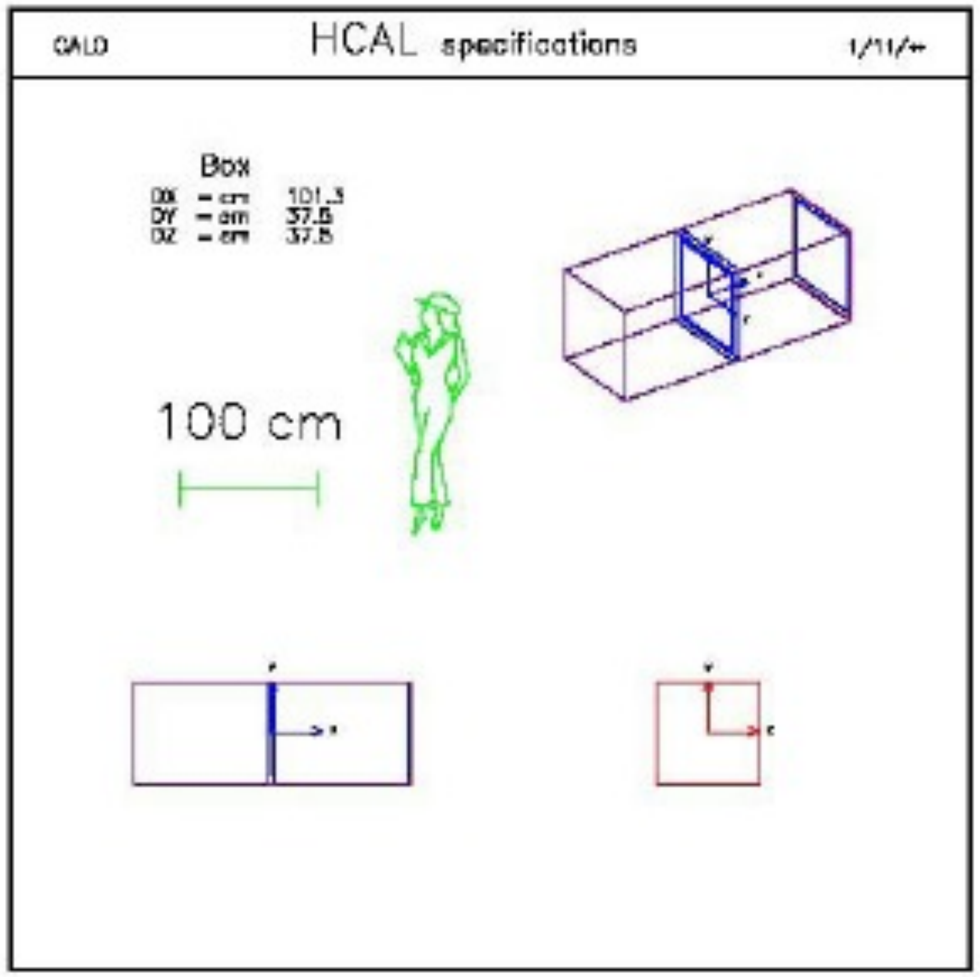


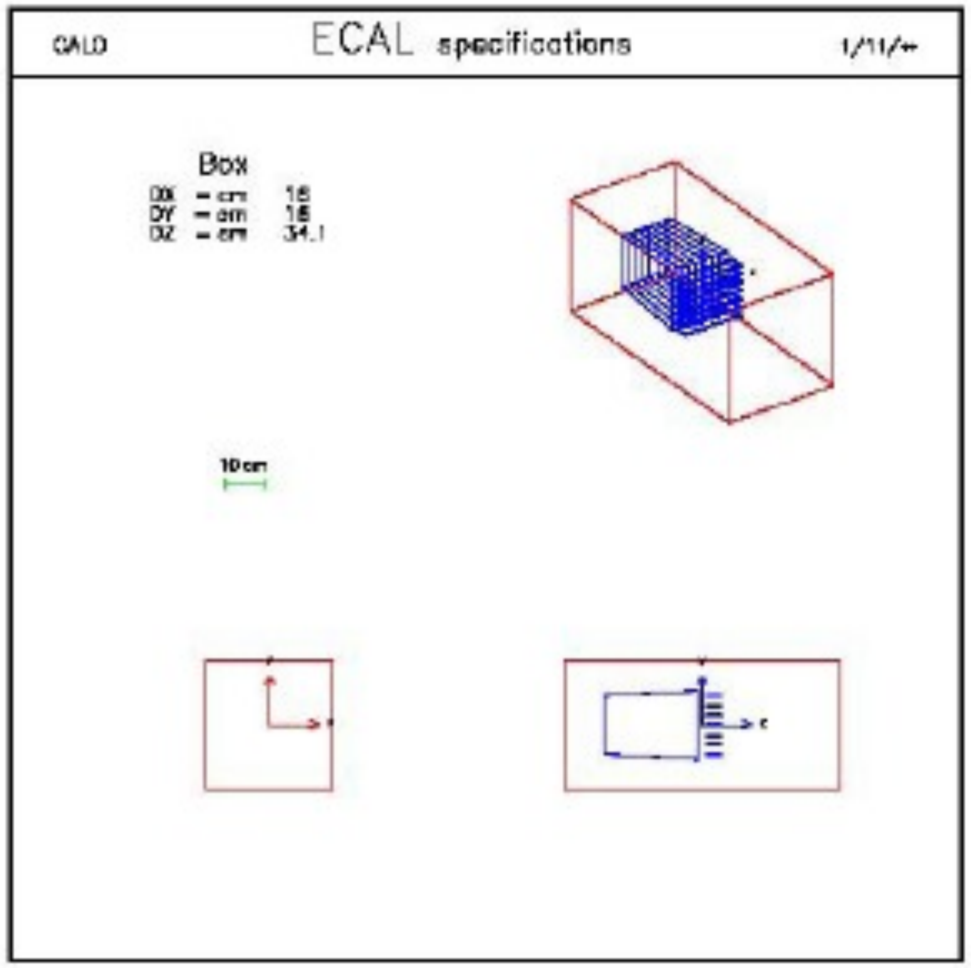
The 1996 CMS Test Beam Setup

- 7 x 7 crystal matrix made out of lead tungstate
- 28 layers of plastic scintillators interleaved with brass plates of varying thickness
- B-field along the z-axis (perpendicular to the beam direction) with maximum field strength of 3 Tesla

Composite-Calorimeter as an Example in Geant4 (advanced)

Composite Calorimeter

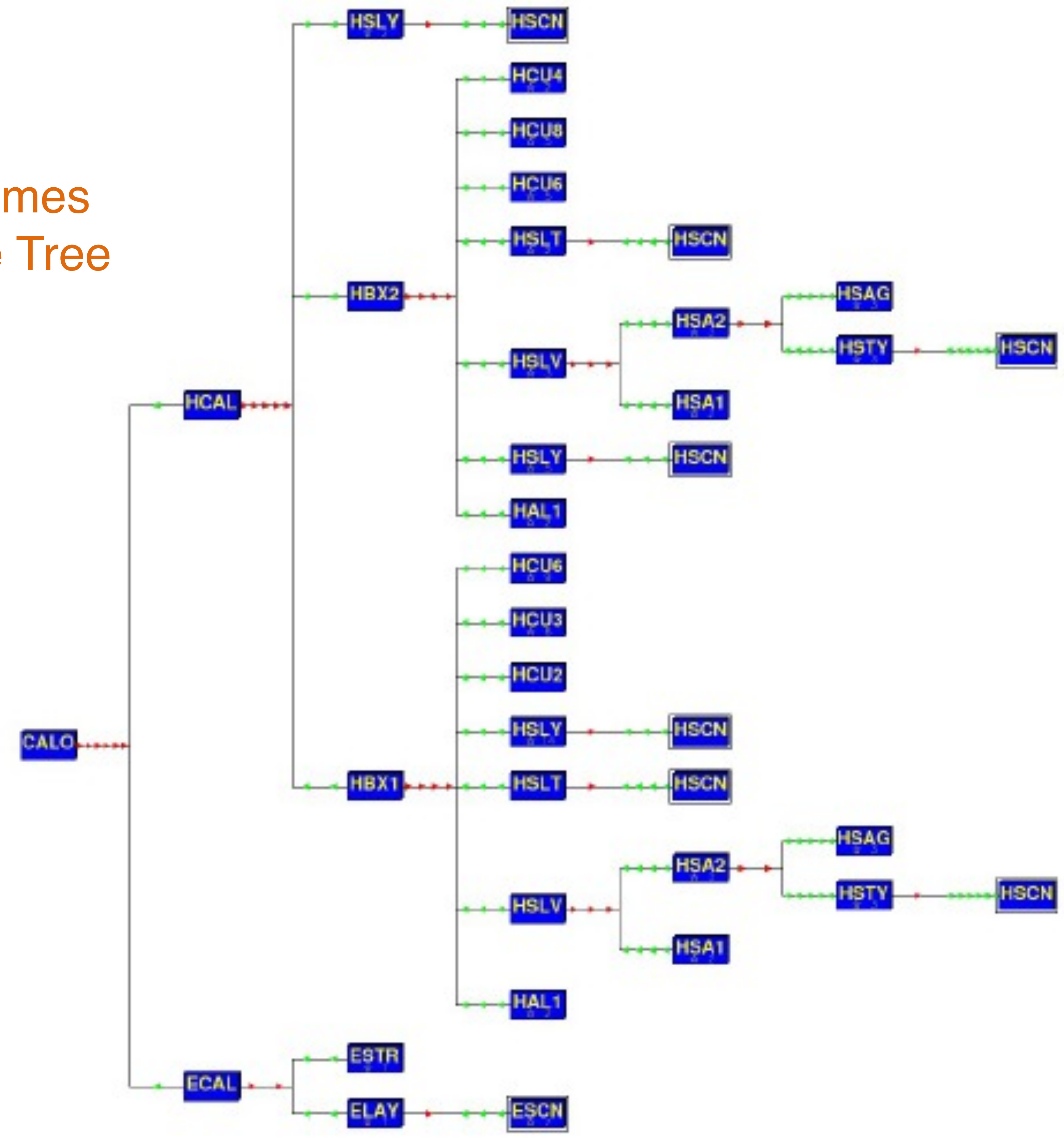




- HCAL (the enclosure for the hadron calorimeter) contains two boxes made out of aluminium each housing absorber plates and scintillator layers
- ECAL (the enclosure for the electromagnetic calorimeter) contains the crystal matrix and some support structure
- Both ECAL and HCAL are placed in a mother volume CALO which defines the world volume

Geometry Tree

22 Logical Volumes
8 Levels in the Tree

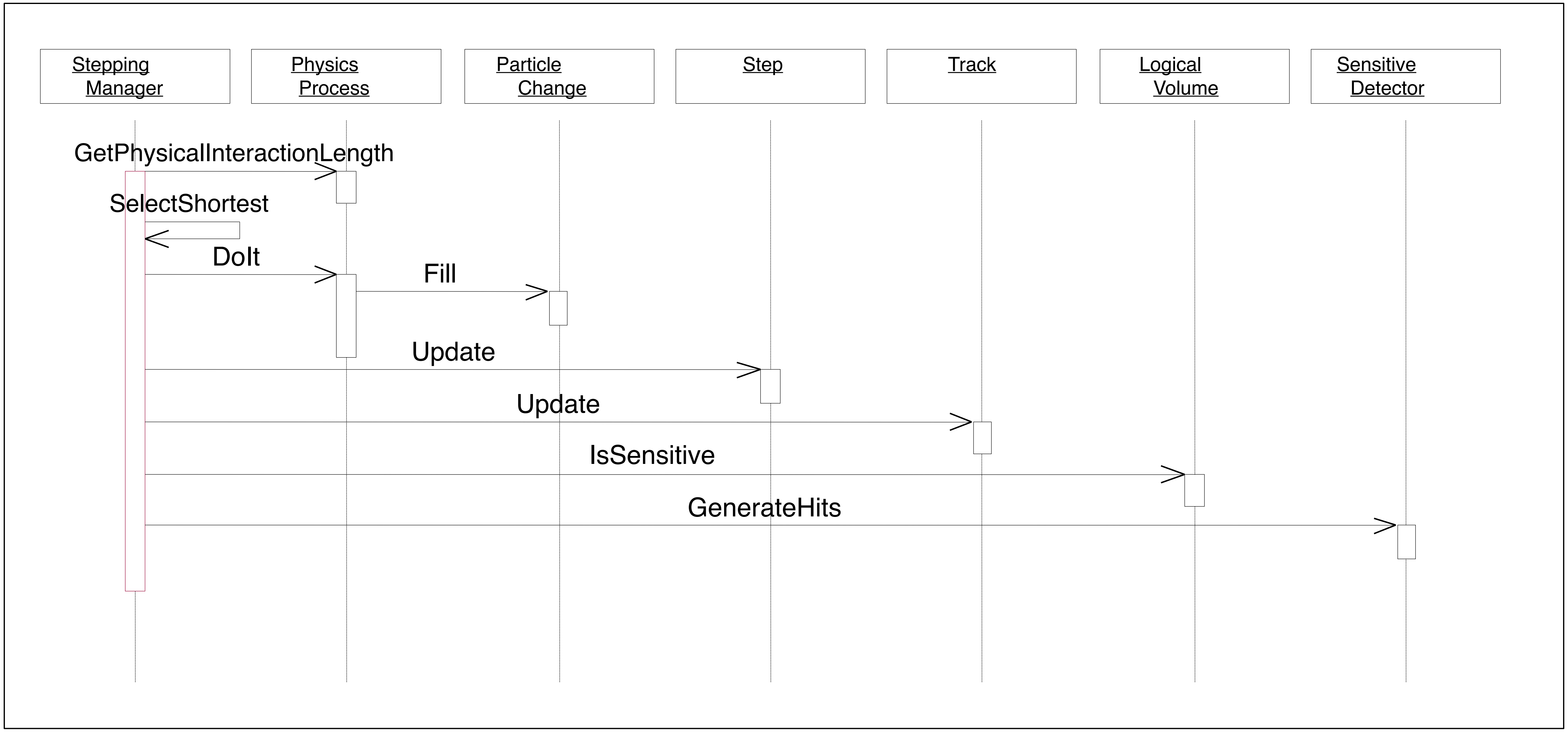


Extraction of Useful information

- Given geometry, physics and primary track generation, Geant4 does proper physics simulation “silently”
  - The user needs to add a bit of code to extract useful information
- There are three ways:
  - Use built-in scoring commands
    - Most commonly-used physics quantities are available
  - Use scorers in the tracking volume
    - Create scores for each event
    - Create own Run class to accumulate scores
  - Assign G4VSensitiveDetector to a volume to generate “hit”
    - Use user hooks (G4UserEventAction, G4UserRunAction) to get event / run summary
- The user may also use user hooks (G4UserTrackingAction, G4UserSteppingAction, etc.)
  - The user has full access to almost all information

Sensitive Detector

- A G4VSensitiveDetector object can be assigned to a G4LogicalVolume
- In case a step takes place in a logical volume that has a G4VSensitiveDetector object, this G4VSensitiveDetector is invoked with the current G4Step object



Defining a Sensitive Detector

- The basic strategy

```
G4LogicalVolume* myLogCalor = ……;
G4VSensetiveDetector* pSensetivePart = new MyDetector("/
  mydet");
G4SDManager* SDMan = G4SDManager::GetSDMpointer();
SDMan->AddNewDetector(pSensitivePart);
myLogCalor->SetSensitiveDetector(pSensetivePart);
```

- Each detector object must have a unique name
  - Some logical volumes can share one detector object
  - More than one detector object can be made from one detector class with different detector name
  - One logical volume cannot have more than one detector object. But, one detector object can generate more than one kind of hits
    - e.g. a double-sided silicon micro-strip detector can generate hits for each side separately

Hit

- Hit is a snapshot of the physical interaction of a Track or an accumulation of interactions of Tracks in the sensitive region of your detector

- A sensitive detector creates Hit(s) using the information given in the G4Step object. The user has to provide his/her own implementation of the detector response

- Hit objects, which are still the user’s class objects, are collected in a G4Event object at the end of an event

Hit Class

- Hit is a user-defined class derived from G4VHit
- The user can store various types of information by implementing one's own concrete Hit class. For example:
  - Position and time of the step
  - Momentum and energy of the track
  - Energy deposition of the step
  - Geometrical information
  - or any combination of above
- Hit objects of a concrete Hit class must be stored in a dedicated collection which is instantiated from G4THitsCollection template class
- The collection is associated with a G4Event object via G4HCofThisEvent
- Hits are accessible as collections:
  - through G4Event at the end of the event
    - to be used for analyzing an event
  - through G4SDManager during processing an event
    - to be used for event filtering

Implementation of Hit class

```
#include "G4VHit.hh"
class MyHit : public G4VHit
{
  public:
      MyHit(some_arguments);
      virtual ~MyHit();
      virtual void Draw();
      virtual void Print();
  private:
      // some data members
  public:
      // some set/get methods
};

#include “G4THitsCollection.hh”
typedef G4THitsCollection<MyHit> MyHitsCollection;
```

Sensitive Detector class

- The sensitive detector is a user-defined class derived from the class G4VSensitiveDetector

```
#include "G4VSensitiveDetector.hh"
#include "MyHit.hh"
class G4Step;
class G4HCofThisEvent;
class MyDetector : public G4VSensitiveDetector
{
  public:
     MyDetector(G4String name);
     virtual ~MyDetector();
     virtual void Initialize(G4HCofThisEvent*HCE);
     virtual G4bool ProcessHits(G4Step*aStep,
                          G4TouchableHistory*ROhist);
     virtual void EndOfEvent(G4HCofThisEvent*HCE);
  private:
     MyHitsCollection * hitsCollection;
     G4int collectionID;
};
```

Types of Hit

- A tracker detector typically generates a hit for every single step of every single (charged) track
  - A tracker hit typically contains
    - Position and time
    - Energy deposition of the step
    - Track identifier
    - Some cell identifier

- A calorimeter detector typically generates a hit for every cell and accumulates energy deposition in each cell for all steps of all tracks
  - A calorimeter hit typically contains
    - Sum of deposited energy
    - Some timing information
    - Cell Identifier

- The user can instantiate more than one object for one sensitive detector class. Each object should have its unique detector name
  - For example, each of the two sets of detectors can have its dedicated sensitive detector objects. But, their functionalities are exactly the same so that they can share the same class. See examples/extended/analysis/A01 as an example

Implementation of Sensitive Detector - 1

```
MyDetector::MyDetector(G4String detector_name)
               :G4VSensitiveDetector(detector_name),
                collectionID(-1)
{
  collectionName.insert(“collection_name");
}
```

- In the constructor, the name of the hits collection which is handled by this sensitive detector is to be defined
- In case the sensitive detector generates more than one kind of hits (e.g. anode and cathode hits separately), all collection names need to be defined

Implementation of Sensitive Detector - 2

```
void MyDetector::Initialize(G4HCofThisEvent*HCE)
{
  if(collectionID<0) collectionID = GetCollectionID(0);
  hitsCollection = new MyHitsCollection
        (SensitiveDetectorName,collectionName[0]);
  HCE->AddHitsCollection(collectionID,hitsCollection);
}
```

- Initialize() method is invoked at the beginning of each event.
- Get the unique ID number for this collection
  - GetCollectionID() is a heavy operation. It should not be used for every event
  - GetCollectionID() is available after this sensitive detector object is constructed and registered to G4SDManager. Thus, this method cannot be invoked in the constructor of this detector class
- The hits collection(s) are to be instantiated and then attached to the G4HCofThisEvent object given in the argument
- In the case of a calorimeter-type detector, hits for all calorimeter cells may be instantiated with zero energy depositions, and then inserted into the collection

Implementation of Sensitive Detector - 3

```
G4bool MyDetector::ProcessHits
  (G4Step*aStep,G4TouchableHistory*ROhist)
{
  MyHit* aHit = new MyHit();
  ...
  // some set methods
  ...
  hitsCollection->insert(aHit);
  return true;
}
```

- This ProcessHits() method is invoked for every step in the volume(s) where this sensitive detector is assigned
- In this method, generate a hit corresponding to the current step (for tracking detector), or accumulate the energy deposition of the current step to the existing hit object where the current step belongs (for calorimeter detector)
- The geometry information is to be collected (e.g. copy number) from "PreStepPoint"
- Currently, the returned boolean value is not used

Implementation of Sensitive Detector - 4

```
void MyDetector::EndOfEvent(G4HCofThisEvent*HCE)  {;}
```

- This method is invoked at the end of processing an event
  - It is invoked even if the event is aborted
  - It is invoked before UserEndOfEventAction

Step Point and Touchable

- As mentioned already, G4Step has two G4StepPoint objects as its starting and ending points. All the geometrical information of the particular step should be taken from "PreStepPoint"
  - Geometrical information associated with G4Track is identical to "PostStepPoint"

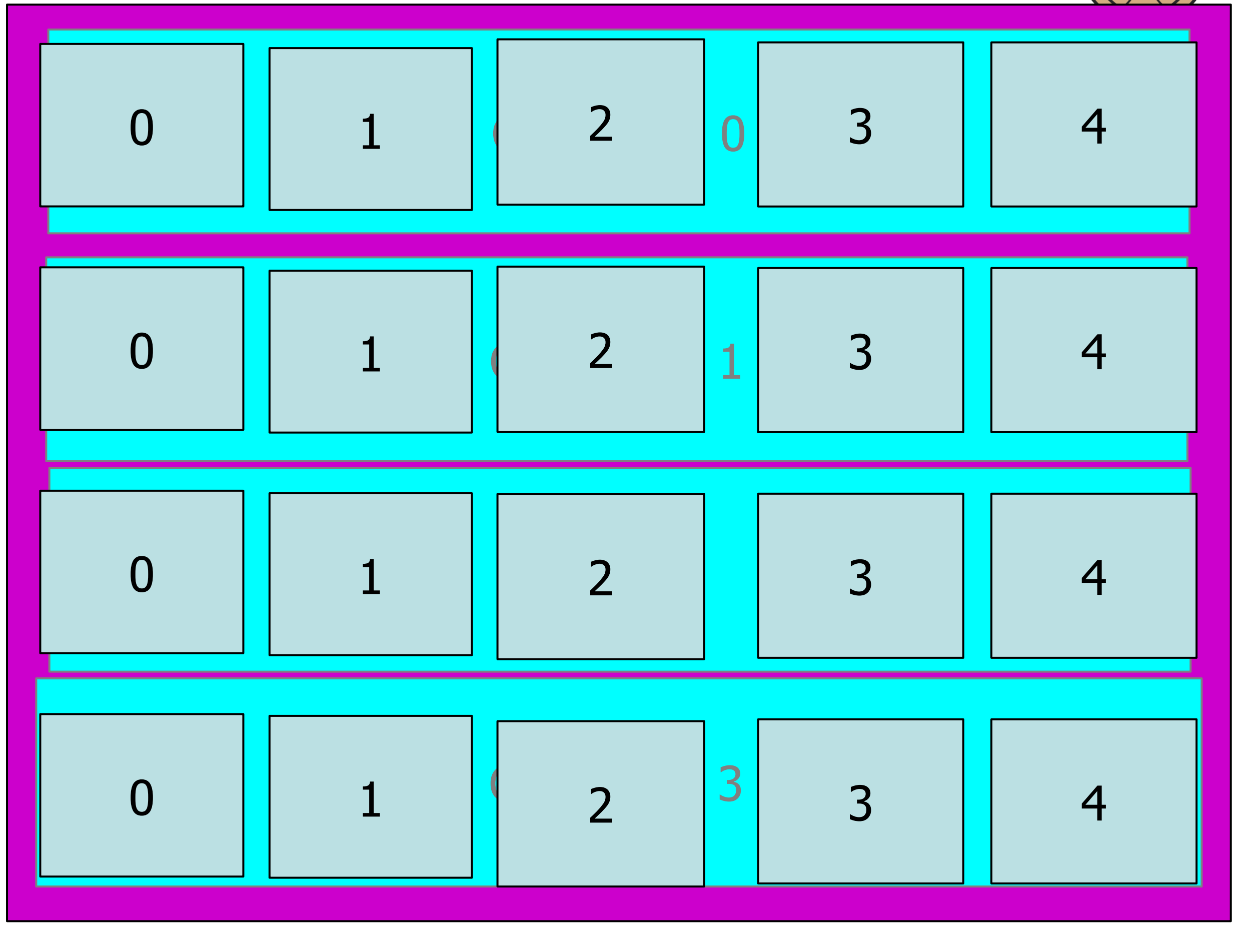
- Each G4StepPoint object has
  - Position in the world coordinate system
  - Global and local time
  - Material
  - G4TouchableHistory for geometrical information

- The G4TouchableHistory object is a vector of information for each geometrical hierarchy
  - copy number
  - translation/rotation to its mother

- Since release 4.0, *handles* (or *smart-pointers*) to touchable are intrinsically used. Touchables are reference counted

Copy Number

- Suppose a calorimeter is made of 4x5 cells
  - and it is implemented by two levels of replica
- In reality, there is only one physical volume object for each level. Its position is parameterized by its copy number
- To get the copy number of each level for the cell when the step belongs to two cells,



  - geometrical information in G4Track is identical to that in “PostStepPoint”
  - the user cannot get the correct copy number for "PreStepPoint" if one directly accesses the physical volume
- The touchable is to be used to get the proper copy number, transform matrix, etc.

Touchable

- G4TouchableHistory has information on the geometrical hierarchy of the point.

```
G4Step* aStep;
G4StepPoint* preStepPoint = aStep->GetPreStepPoint();
G4TouchableHistory* theTouchable =
   (G4TouchableHistory*)(preStepPoint->GetTouchable());
G4int copyNo = theTouchable->GetVolume()->GetCopyNo();
G4int motherCopyNo
          = theTouchable->GetVolume(1)->GetCopyNo();
G4int grandMotherCopyNo
          = theTouchable->GetVolume(2)->GetCopyNo();
G4ThreeVector worldPos = preStepPoint->GetPosition();
G4ThreeVector localPos = theTouchable->GetHistory()
   ->GetTopTransform().TransformPoint(worldPos);
```

G4HCofThisEvent

- A G4Event object has a G4HCofThisEvent object at the end of (successful) event processing. G4HCofThisEvent object stores all hits collections made within the event.

  - Pointer(s) to the collections may be NULL if collections are not created in the particular event

  - Hit collections are stored by pointers of the G4VHitsCollection base class. Thus, one has to cast them into types of individual concrete classes

  - The index number of a Hits collection is unique and unchanged for a run. The index number can be obtained by

    ```
    G4SDManager::GetCollectionID("detName/colName");
    ```

    - The index table is also stored in G4Run

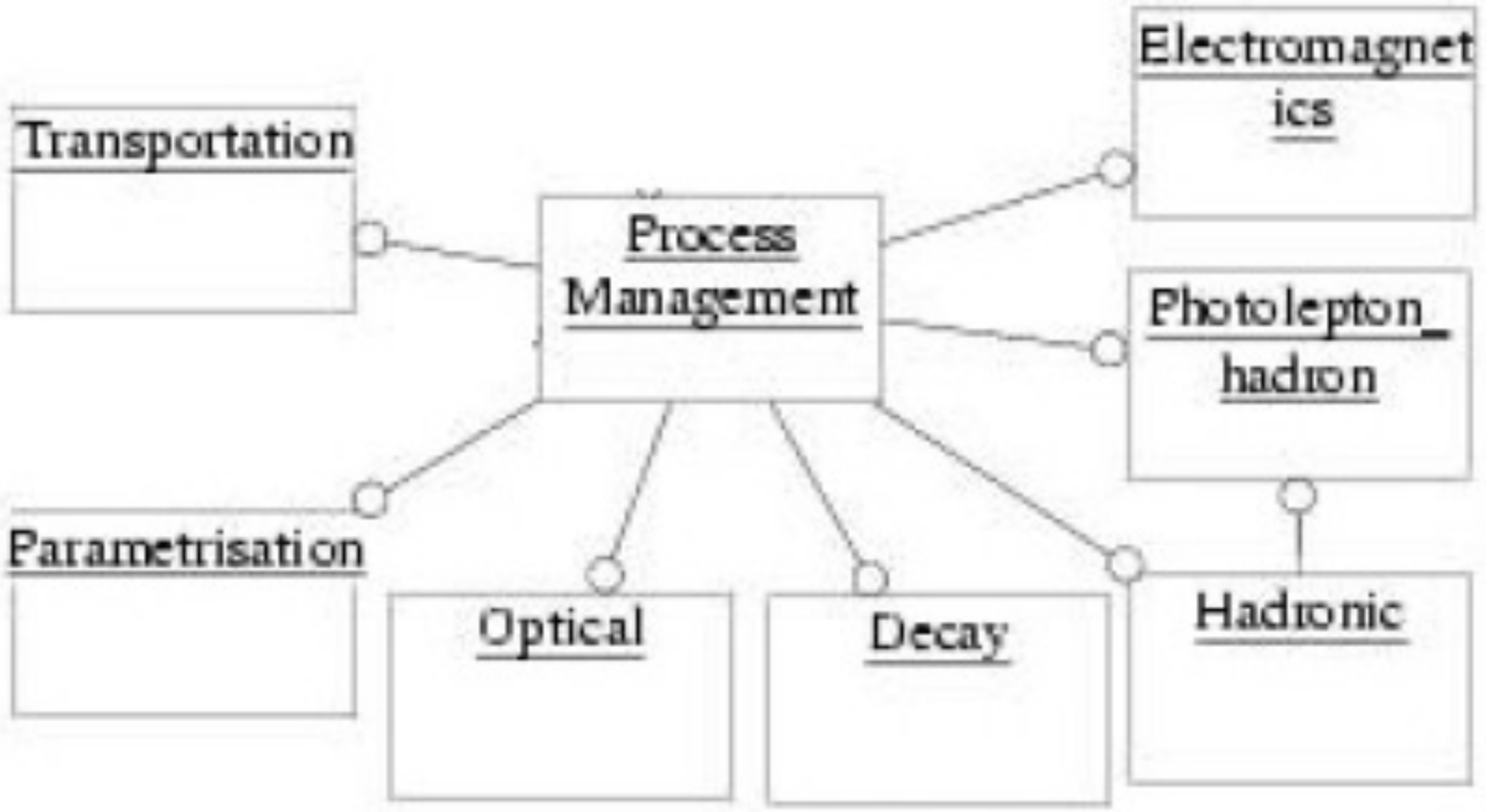
Usage of G4HCofThisEvent

```
void MyEventAction::EndOfEventAction(const G4Event* evt)  {
  static int CHCID = -1;
  If(CHCID<0) CHCID = G4SDManager::GetSDMpointer()
     ->GetCollectionID("myDet/collection1");
  G4HCofThisEvent* HCE = evt->GetHCofThisEvent();
  MyHitsCollection* CHC = 0;
  if (HCE) {
     CHC = (MyHitsCollection*)(HCE->GetHC(CHCID)); }
  if (CHC) {
    int n_hit = CHC->entries();
    G4cout<<"My detector has "<<n_hit<<" hits."<<G4endl;
    for (int i1=0;i1<n_hit;i1++) {
      MyHit* aHit = (*CHC)[i1];
      aHit->Print();
    }
  }
}
```

Physics Lists

- Geant4 provides the possibility of simulating the physics processes of a variety of particles
  - each such particle can have interactions of different types: strong, electromagnetic or weak and each such interaction can be described by different models
  - unlike many other simulation tools, Geant4 leaves it to the user to decide on which particles, which interactions and which models are to be used during the simulation step
  - declaration of the list of particles and the choice of models is done using the physics list
  - the toolkit provides handles for a few well-defined physics lists which are suitable for certain specific types of application

Processes in Geant4



- Processes describe how particles interact with material or with a volume

- Three basic types:
  - At rest process (e.g. decay at rest)
  - Continuous process (e.g. ionisation)
  - Discrete process (e.g. Compton scattering)

- Transportation is also a process
  - Interacting with the volume boundary

- A process which requires the shortest interaction length limits the step

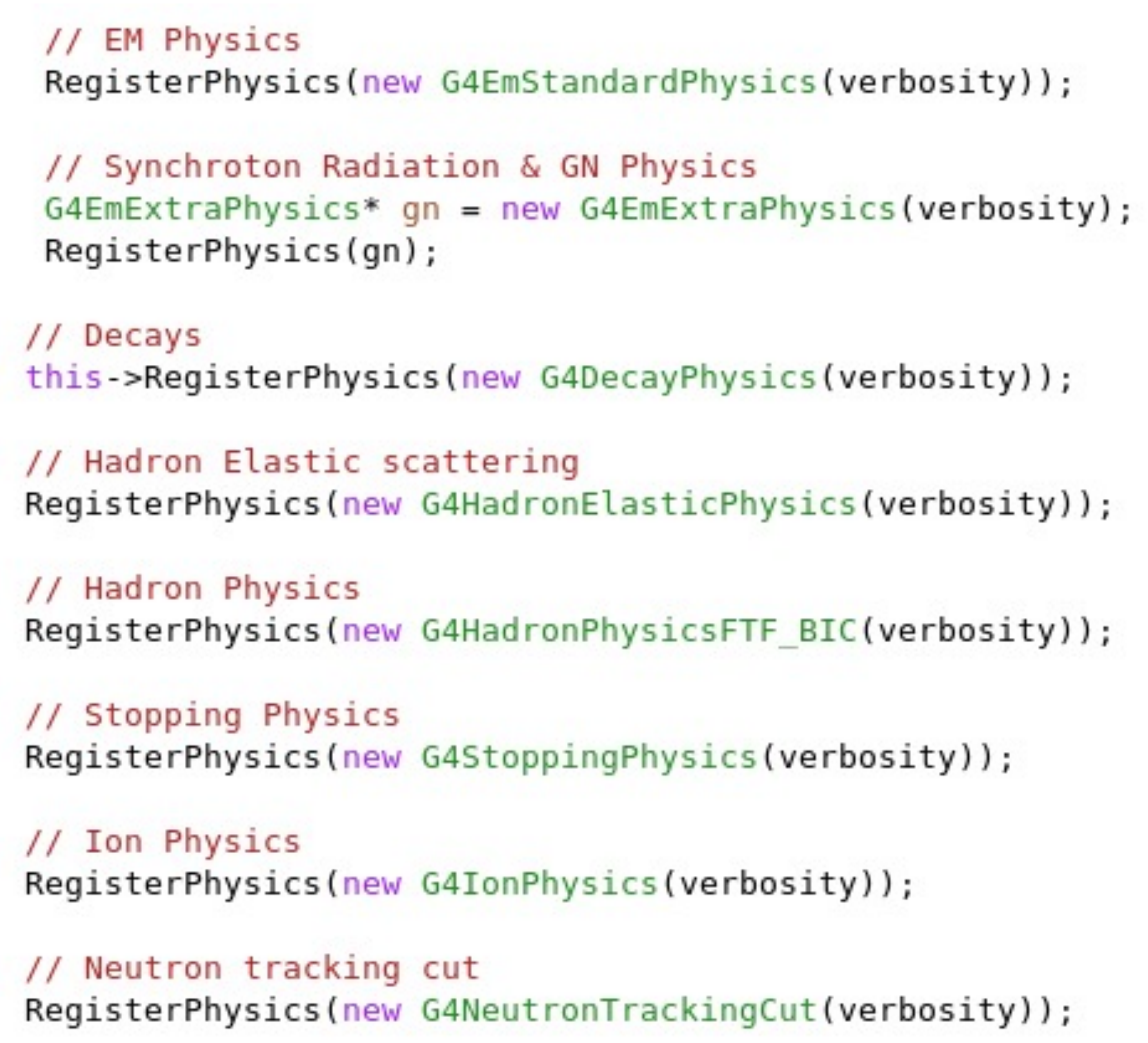
Types of Processes

- There are several types of modules which can be combined to define the physics list
  - electromagnetic physics
  - extra physics processes for photons and leptons
  - decays
  - hadronic elastic
  - hadronic inelastic
  - stopping particles and capture processes
  - ion nuclear interactions
  - step limits
  - others

- The others category includes optical photons, exotic physics processes, thermal neutron transport models, ….

- The user needs to define the physics list through the 3 handles:
  - RegisterPhysics
  - ReplacePhysics
  - RemovePhysics

An Example of Making a Physics List



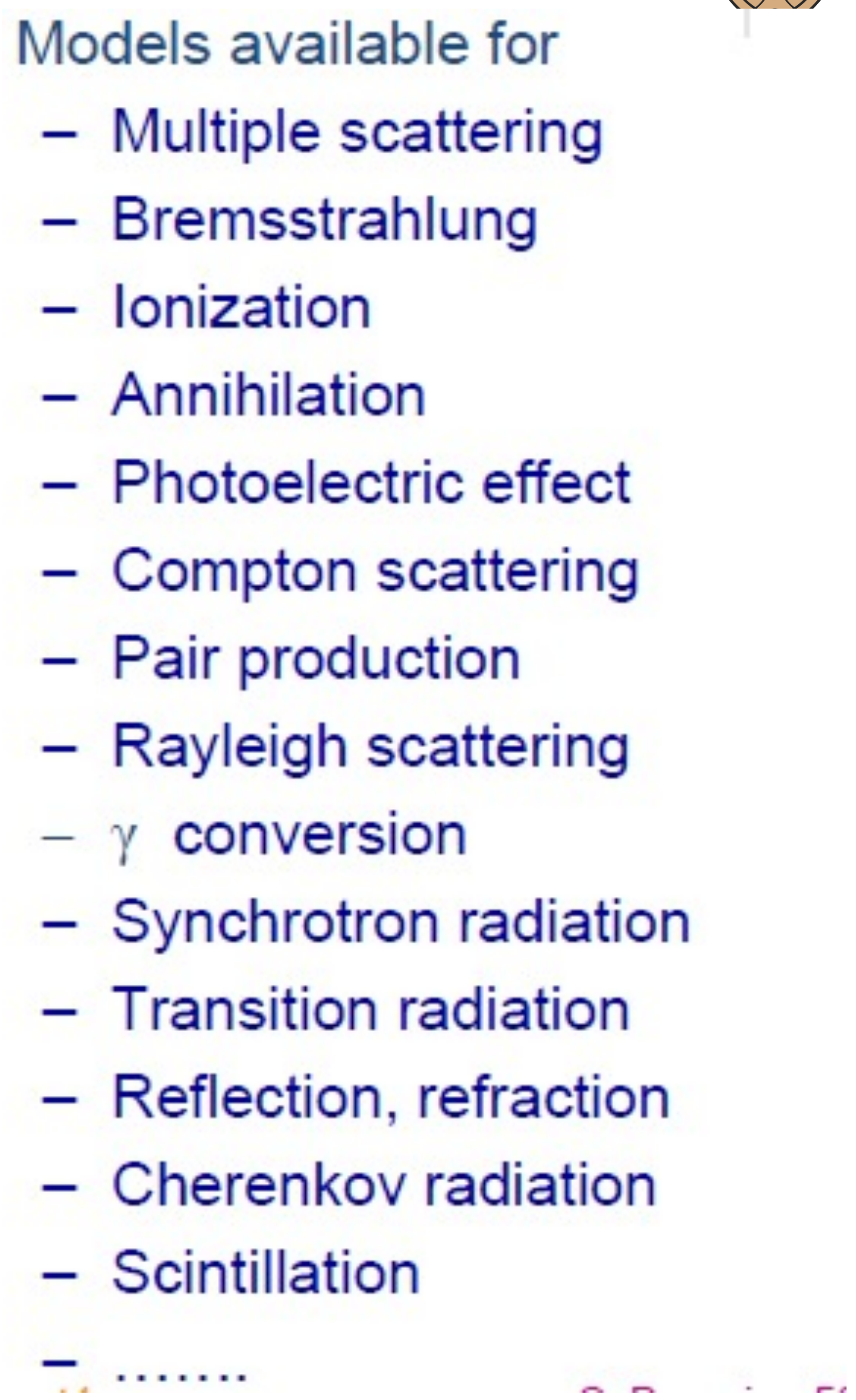
```
 // EM Physics
 RegisterPhysics(new G4EmStandardPhysics(verbosity));

 // Synchroton Radiation & GN Physics
 G4EmExtraPhysics* gn = new G4EmExtraPhysics(verbosity);
 RegisterPhysics(gn);

// Decays
this->RegisterPhysics(new G4DecayPhysics(verbosity));

// Hadron Elastic scattering
RegisterPhysics(new G4HadronElasticPhysics(verbosity));

// Hadron Physics
RegisterPhysics(new G4HadronPhysicsFTF_BIC(verbosity));

// Stopping Physics
RegisterPhysics(new G4StoppingPhysics(verbosity));

// Ion Physics
RegisterPhysics(new G4IonPhysics(verbosity));

// Neutron tracking cut
RegisterPhysics(new G4NeutronTrackingCut(verbosity));
```

Electromagnetic Physics

- Applicable to
  - electrons and positrons
  - γ, X-ray and optical photons
  - muons
  - charged hadrons
  - ions

- Several physics models are available. Standard EM physics is extended to low energies using many data-driven techniques to improve the quality of simulation at low energies

- *All obey the same abstract Process interface*: transparent to tracking



Models available for

- Multiple scattering
- Bremsstrahlung
- Ionization
- Annihilation
- Photoelectric effect
- Compton scattering
- Pair production
- Rayleigh scattering
- γ conversion
- Synchrotron radiation
- Transition radiation
- Reflection, refraction
- Cherenkov radiation
- Scintillation
- ....

Geant4 Options for EM Physics

- There are 10+ options for EM physics
  - opt1 (EMV): a fast but less precise version used in HE physics lists
  - opt2 (EMX): also a fast and less precise version for HE physics lists
  - opt3 (EMY): provides a more accurate simulation of photons and charged hadrons
  - opt4 (EMZ): most precise but slow description of EM physics for HE applications
  - LIV: similar to opt3 but models for photos and electrons make use of Livermore set of models
  - PEN: similar to opt3 with models for photons and electrons making use of Penelope set of models
  - _GS: substitute Urban multiple scattering models with the Goudschmidt-Sanderson model
  - _LE: low energy WentzelVI model is used for multiple scattering
  - WVI: WentzelVI model and ATIMA ion ionisation models are used for a better description of multiple scattering
  - _SS: single scattering models used on top of standard EM configuration

- Please note that the same model describes EM physics over the entire energy region

Hadronic Processes

- Hadronic processes are often implemented in terms of a model class

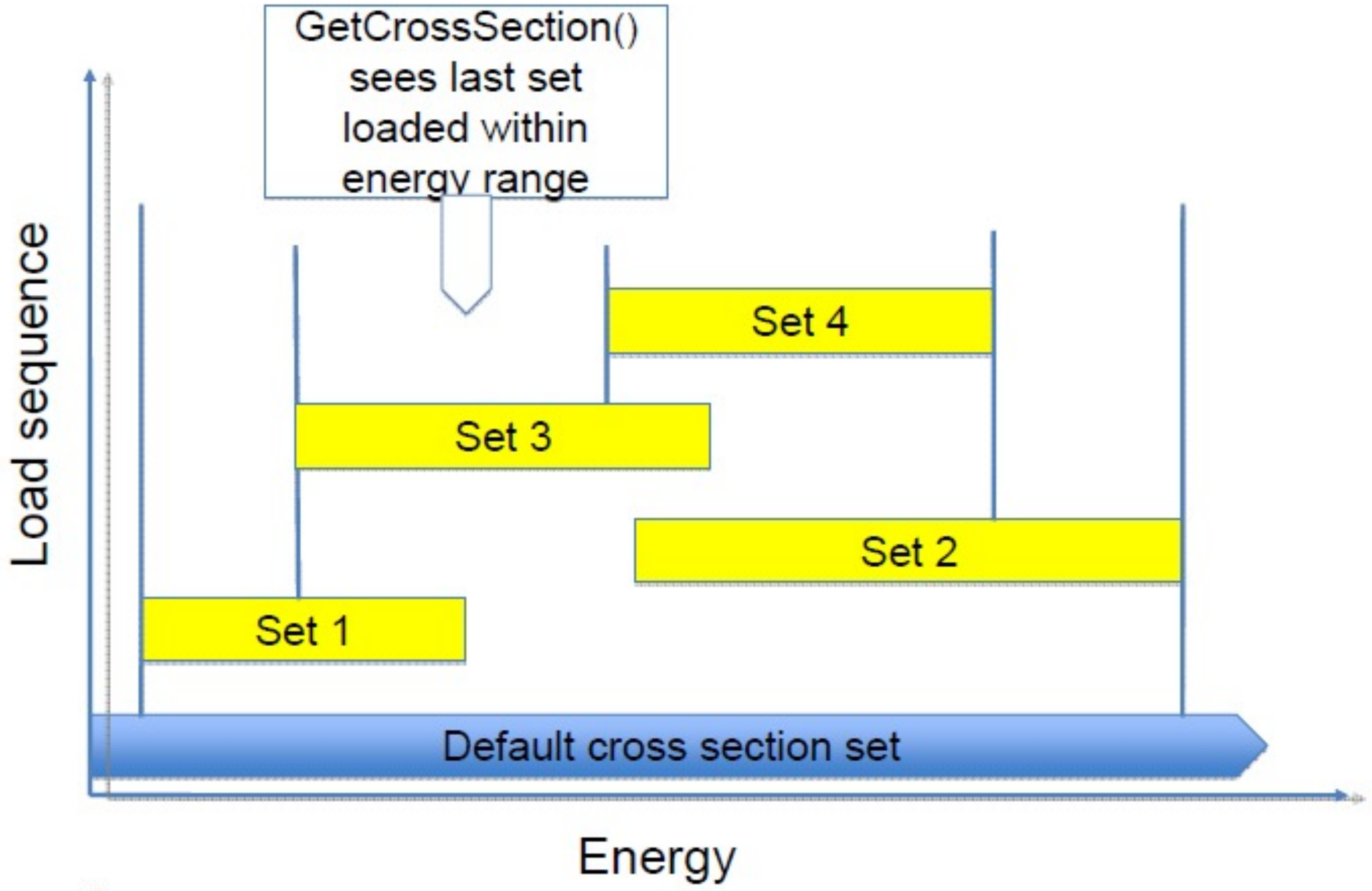
- There are usually several models for a given process
  - user must choose
  - can, and sometimes must, have more than one per process

- A process must also have cross sections assigned
  - here too, there are options

- Default cross-section sets are provided for each type of hadronic process
  - fission, capture, elastic, inelastic
  - can be overridden or completely replaced

- Different types of cross-section sets exist
  - some contain a few numbers to parametric the cross-section as a function of energy
  - some represent large databases
  - some are purely theoretical (equation driven)

Cross Section Management


GetCrossSection() sees last set loaded within energy range
Set 4
Set 3
Set 2
Set 1
Default cross section set
Load sequence
Energy

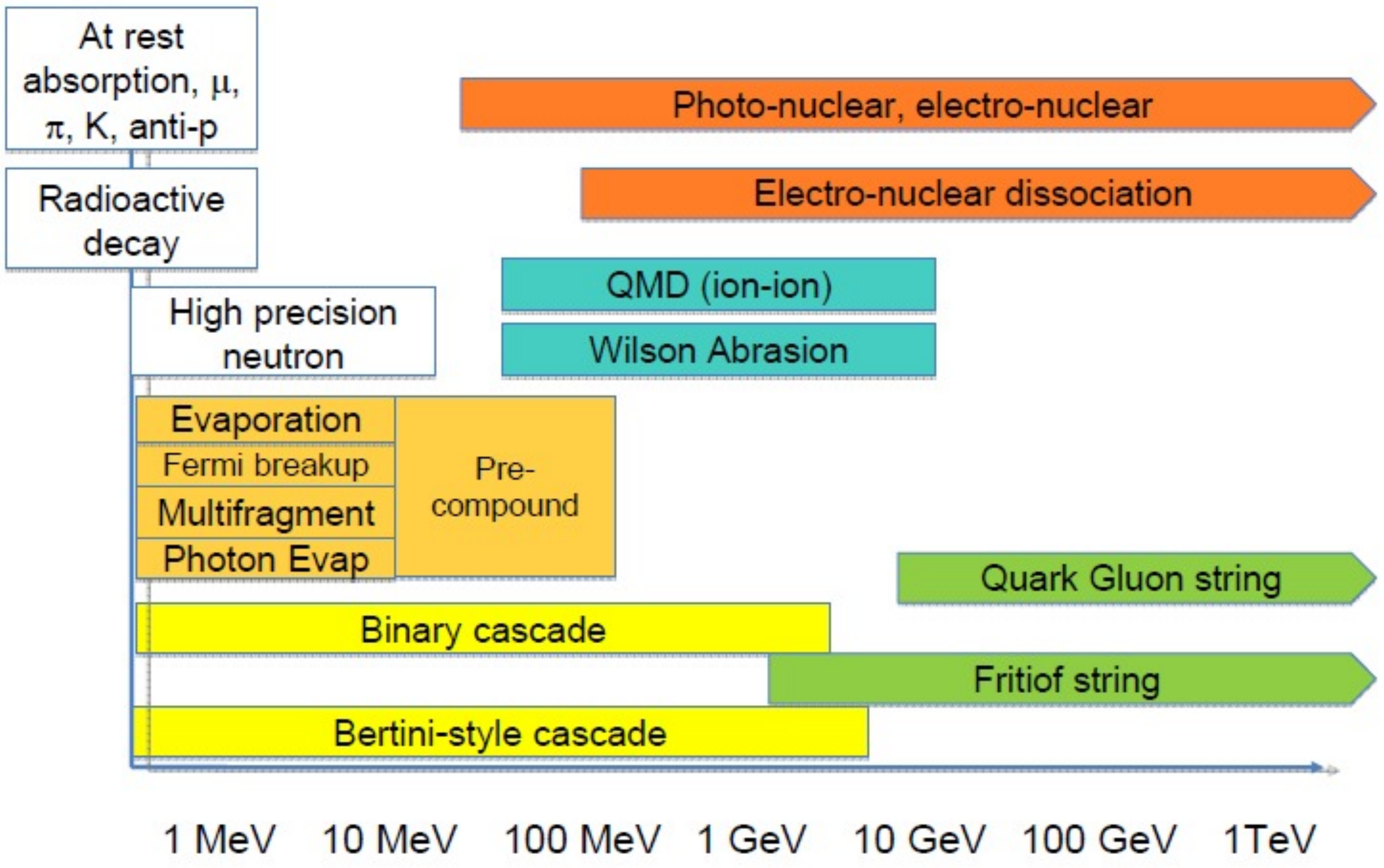
Models in Hadronic Interactions

- Data-driven models: When sufficient data are available with sufficient coverage over the phase space, a data-driven approach is the optimal way
  - neutron transport, photon evaporation, absorption at rest, isotope production, inclusive cross section, ……

- Parametrised models: Extrapolation of cross sections and parametrisation of multiplicity distributions and final state kinematics
  - adaptation of GHEISHA in some earlier versions of Geant4

- Theory-based models: These include a set of theoretical models describing hadronic interactions depending on the addressed energy range
  - diffractive string excitation, dual parton model or quark-gluon string model at medium to high energies
  - intra-nuclear cascade models air medium to low energies
  - nuclear evaporation, fission models, …… at very low energies

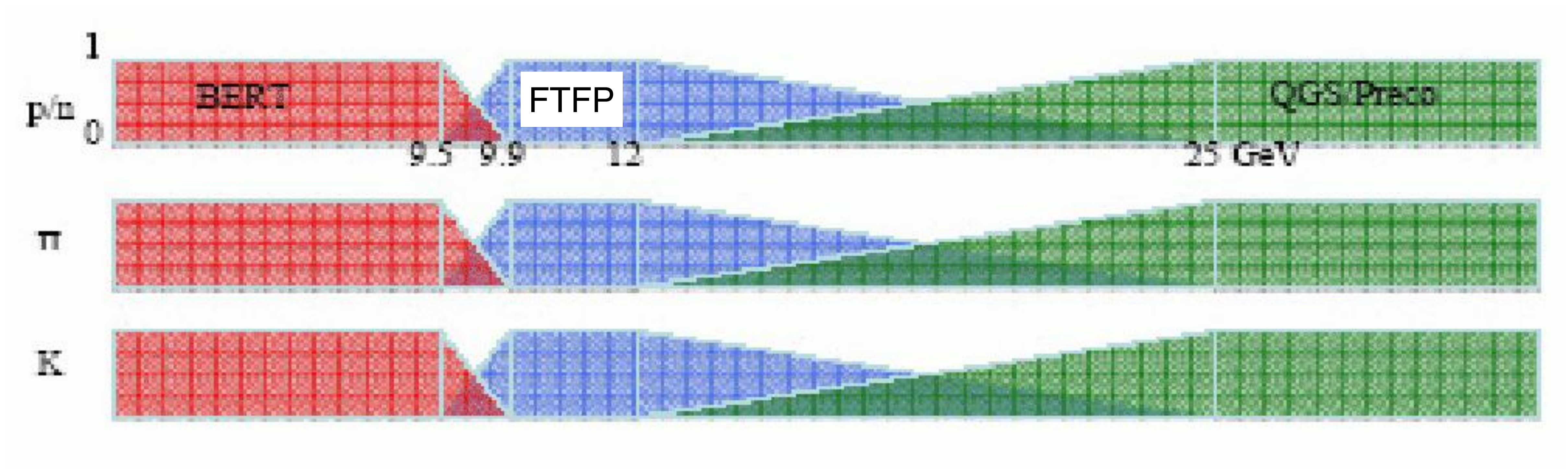
Partial Hadronic Model Inventory



Inelastic Hadronic Interactions

- No single model for hadron inelastic process can cover the entire energy region required in a high-energy physics experiment
  - Quark-gluon string models are good at high energies
  - Nuclear cascade models are good at medium and low energies
  - At very low energies, models for fission and pre-combination are required

- So all physics lists for inelastic hadronic interaction combine a number of models



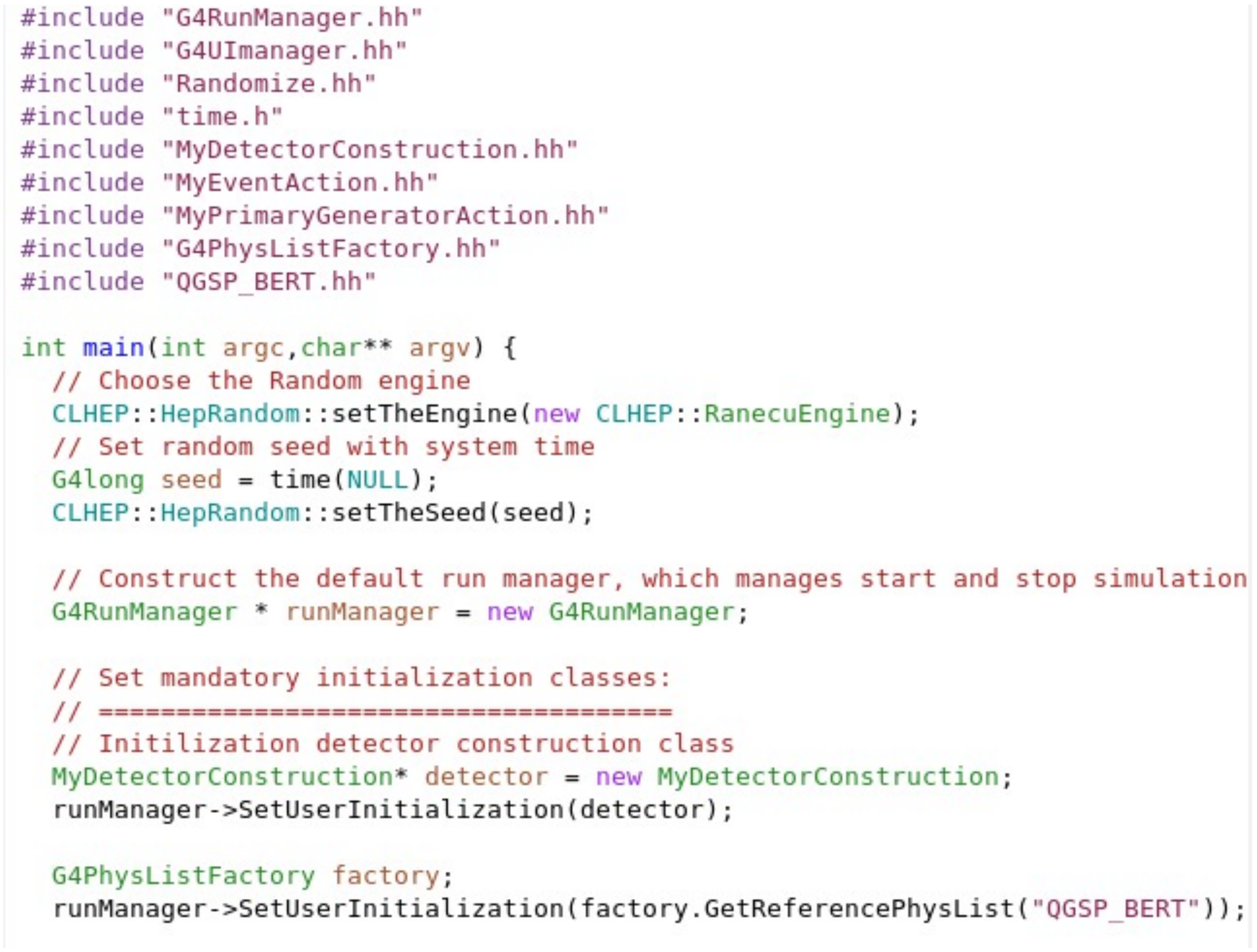
Physics List Library

- There are a number of Physics Lists available in the Geant4 library which can be directly incorporated into the user code

• FTFP_BERT	High Energy Experiments
• FTFP_BERT_ATL	
• FTFP_BERT_HP	
• FTFP_BERT_TRV	
• FTFP_INCLXX	
• FTFQGSP_BERT	
• FTF_BIC	
• QBBC	Medical and Space Physics Applications
• QGSP_BERT	Former default for High Energy Experiments
• QGSP_BERT_HP	
• QGSP_BIC	Cosmic Ray applications
• QGSP_BIC_AllHP	
• QGSP_BIC_HIP	
• QGSP_FTFP_BERT	
• QGSP_INCLXX	
• QGS_BIC	
• Shielding	Recommended for shielding studies
• ShieldingLEND	
• LBE	
• NuBeam	

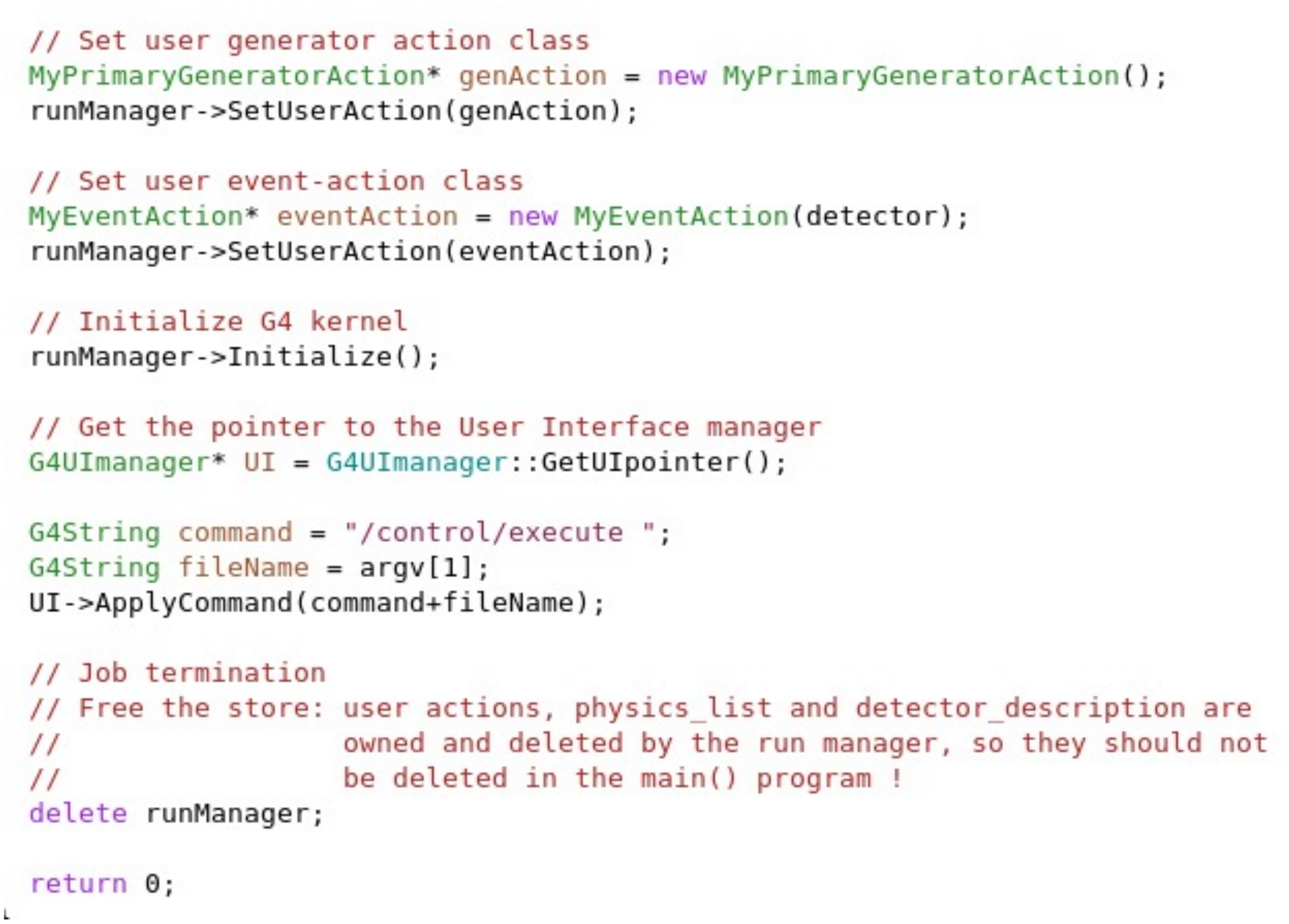
Main Program

- Simple example of the main program:



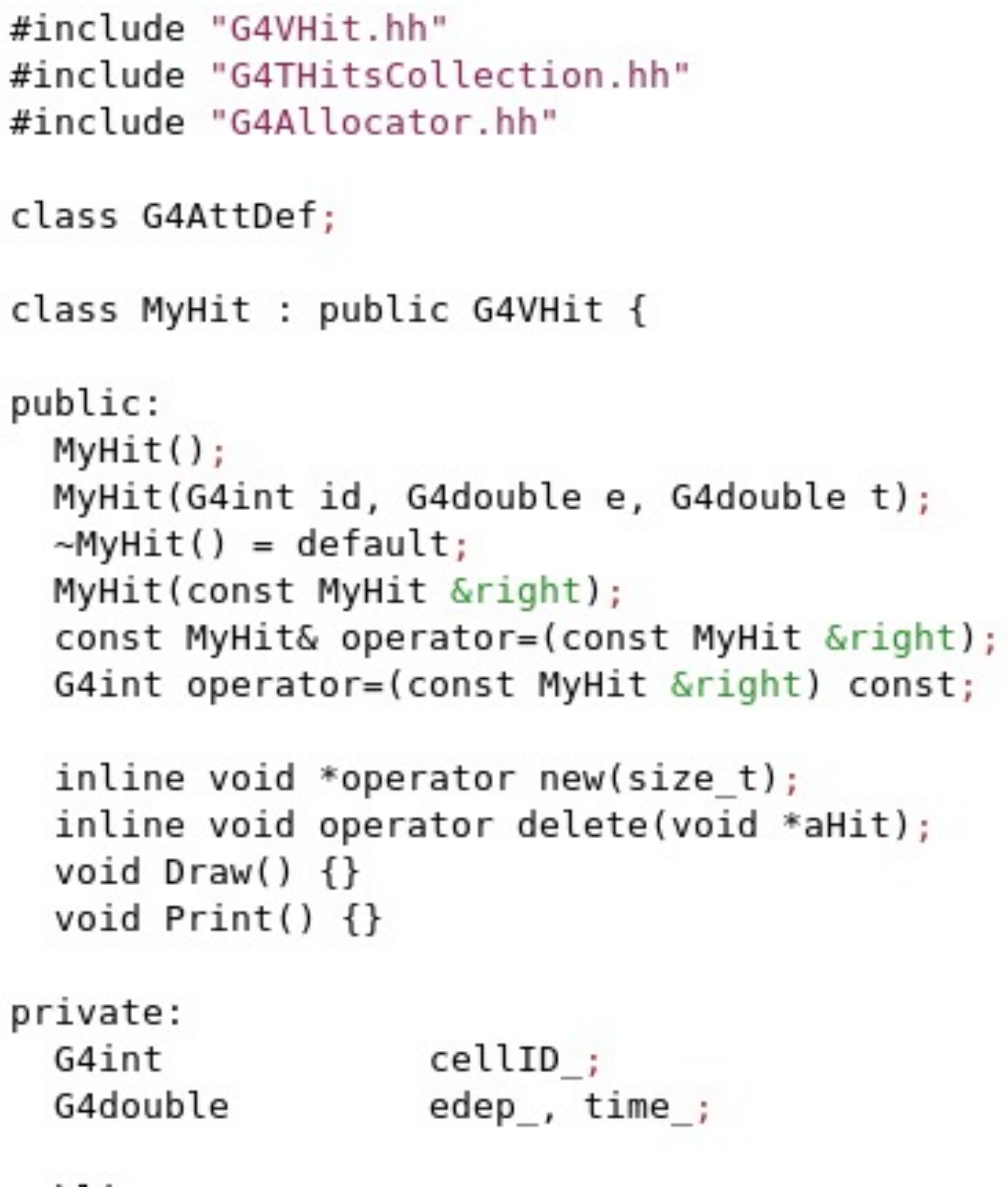
```
#include "G4RunManager.hh"
#include "G4UImanager.hh"
#include "Randomize.hh"
#include "time.h"
#include "MyDetectorConstruction.hh"
#include "MyEventAction.hh"
#include "MyPrimaryGeneratorAction.hh"
#include "G4PhysListFactory.hh"
#include "QGSP_BERT.hh"

int main(int argc,char** argv) {
  // Choose the Random engine
  CLHEP::HepRandom::setTheEngine(new CLHEP::RanecuEngine);
  // Set random seed with system time
  G4long seed = time(NULL);
  CLHEP::HepRandom::setTheSeed(seed);

  // Construct the default run manager, which manages start and stop simulation
  G4RunManager * runManager = new G4RunManager;

  // Set mandatory initialization classes:
  // =====================================
  // Initilization detector construction class
  MyDetectorConstruction* detector = new MyDetectorConstruction;
  runManager->SetUserInitialization(detector);

  G4PhysListFactory factory;
  runManager->SetUserInitialization(factory.GetReferencePhysList("QGSP_BERT"));
```

Main Program (cont)

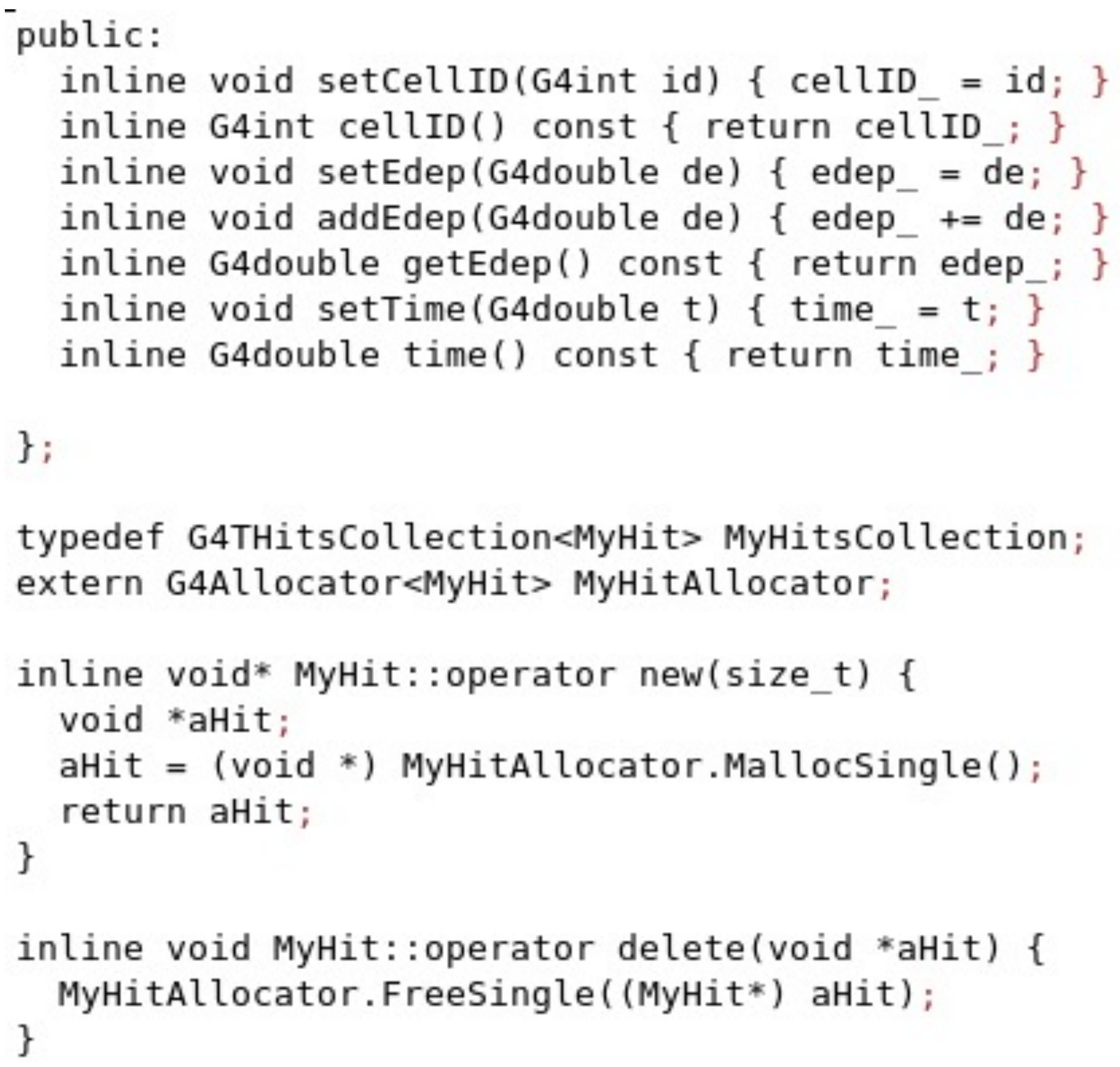


```
// Set user generator action class
MyPrimaryGeneratorAction* genAction = new MyPrimaryGeneratorAction();
runManager->SetUserAction(genAction);

// Set user event-action class
MyEventAction* eventAction = new MyEventAction(detector);
runManager->SetUserAction(eventAction);

// Initialize G4 kernel
runManager->Initialize();

// Get the pointer to the User Interface manager
G4UImanager* UI = G4UImanager::GetUIpointer();

G4String command = "/control/execute ";
G4String fileName = argv[1];
UI->ApplyCommand(command+fileName);

// Job termination
// Free the store: user actions, physics_list and detector_description are
//                 owned and deleted by the run manager, so they should not
//                 be deleted in the main() program !
delete runManager;

return 0;
```

Hit Class

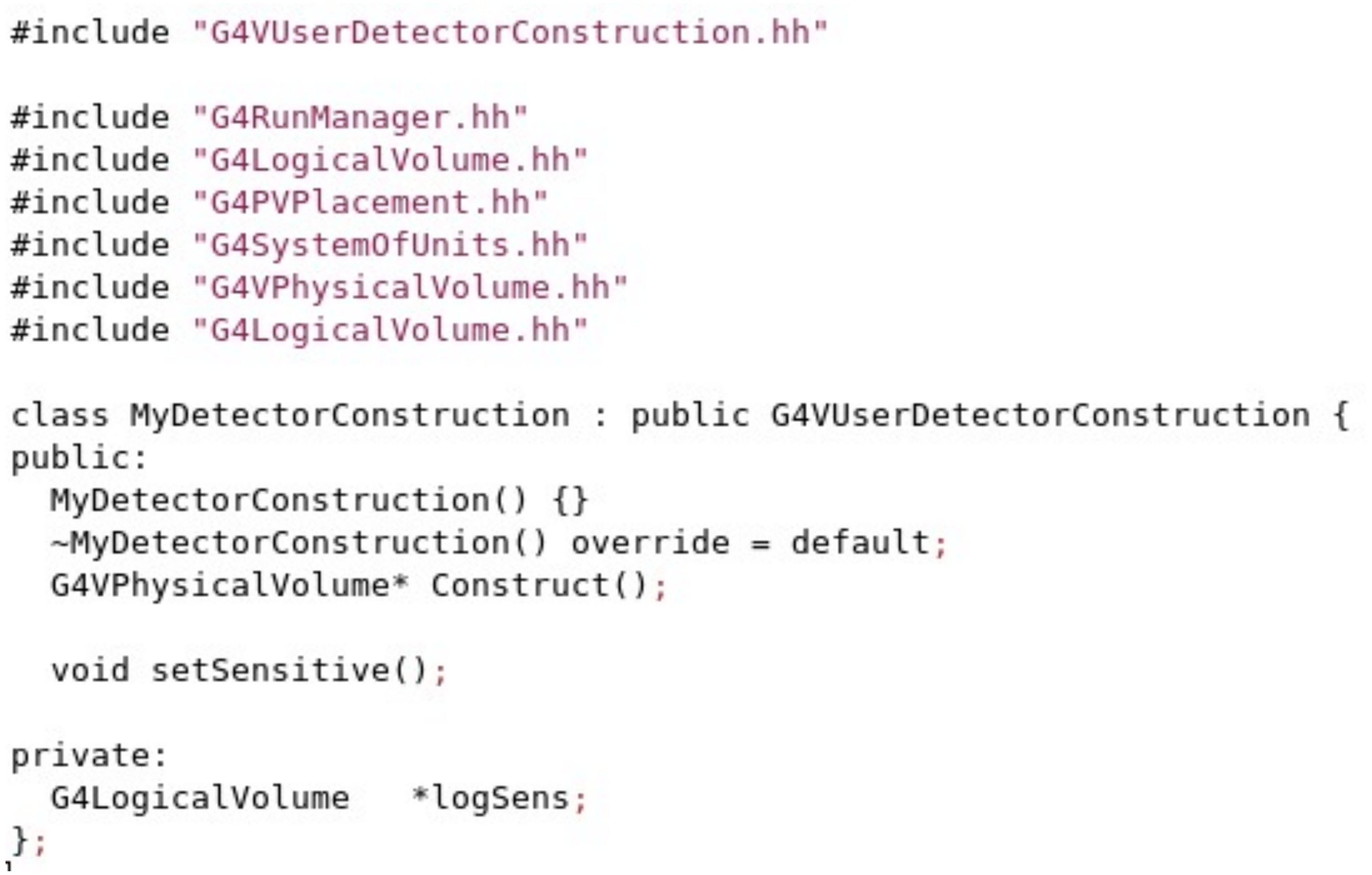


```
#include "G4VHit.hh"
#include "G4THitsCollection.hh"
#include "G4Allocator.hh"

class G4AttDef;

class MyHit : public G4VHit {

public:
  MyHit();
  MyHit(G4int id, G4double e, G4double t);
  ~MyHit() = default;
  MyHit(const MyHit &right);
  const MyHit& operator=(const MyHit &right);
  G4int operator=(const MyHit &right) const;

  inline void *operator new(size_t);
  inline void operator delete(void *aHit);
  void Draw() {}
  void Print() {}

private:
  G4int             cellID_;
  G4double          edep_, time_;

  ...
```



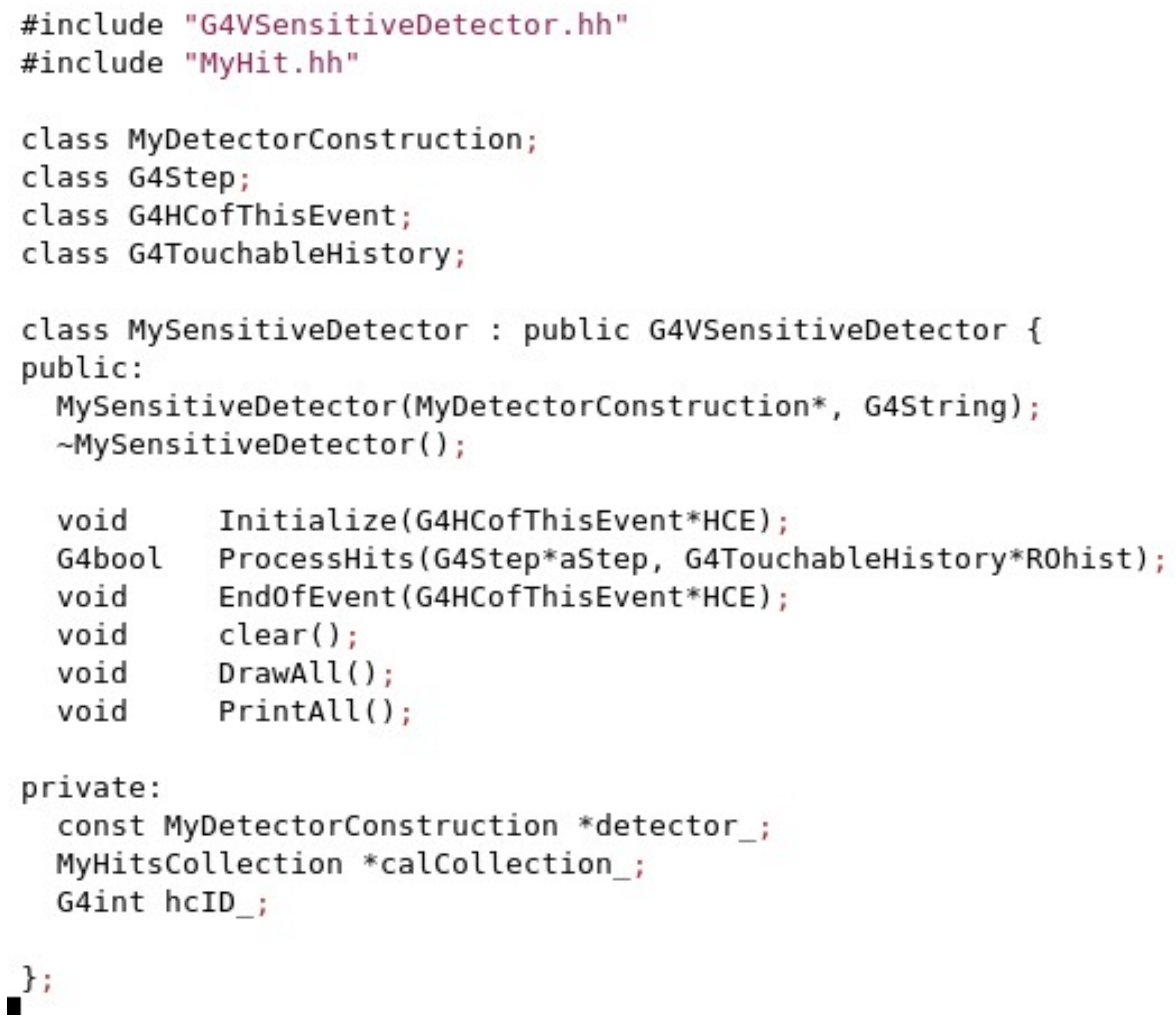
```
public:
  inline void setCellID(G4int id) { cellID_ = id; }
  inline G4int cellID() const { return cellID_; }
  inline void setEdep(G4double de) { edep_ = de; }
  inline void addEdep(G4double de) { edep_ += de; }
  inline G4double getEdep() const { return edep_; }
  inline void setTime(G4double t) { time_ = t; }
  inline G4double time() const { return time_; }

};

typedef G4THitsCollection<MyHit> MyHitsCollection;
extern G4Allocator<MyHit> MyHitAllocator;

inline void* MyHit::operator new(size_t) {
  void *aHit;
  aHit = (void *) MyHitAllocator.MallocSingle();
  return aHit;
}

inline void MyHit::operator delete(void *aHit) {
  MyHitAllocator.FreeSingle((MyHit*) aHit);
}
```

Detector Construction



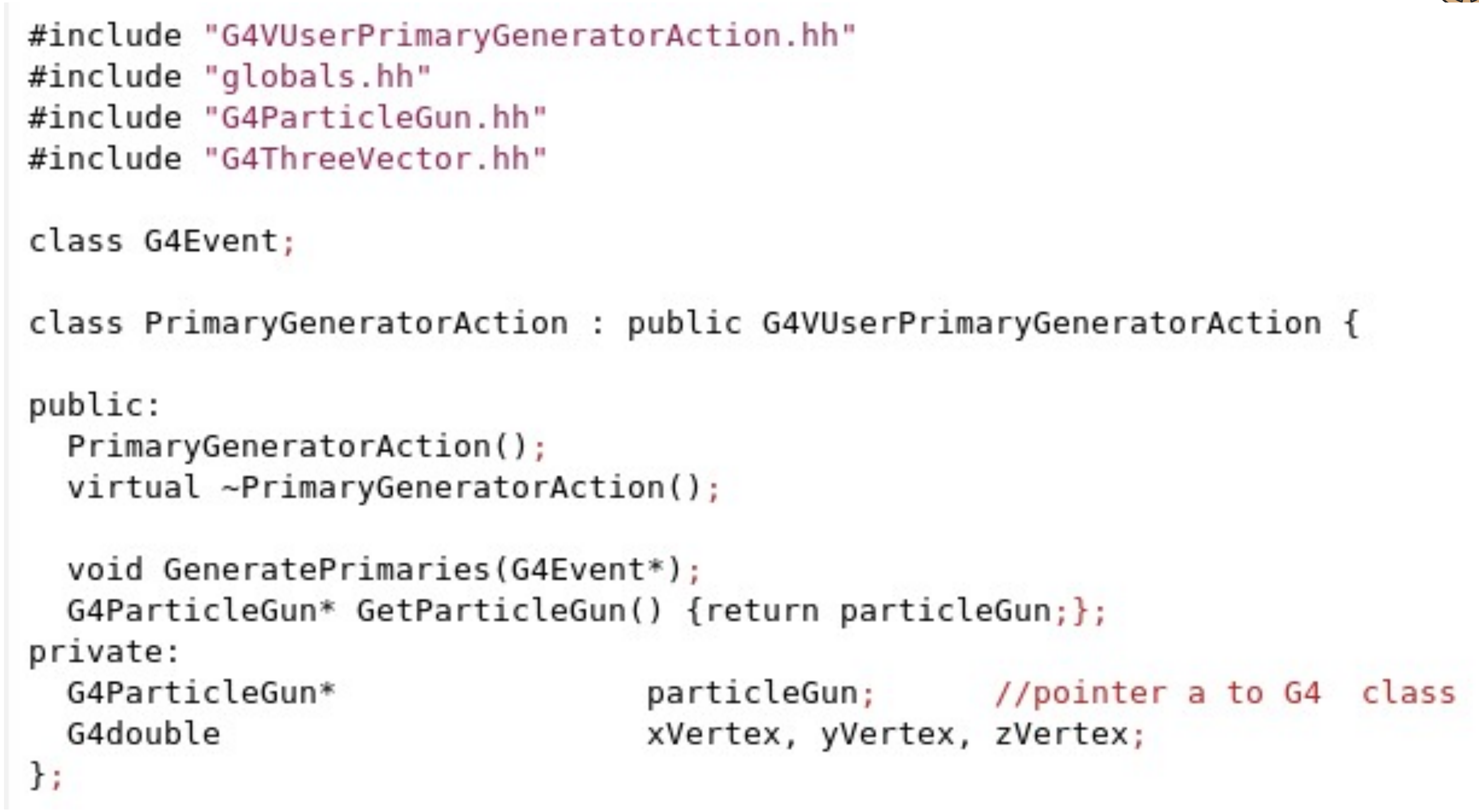
```
#include "G4VUserDetectorConstruction.hh"

#include "G4RunManager.hh"
#include "G4LogicalVolume.hh"
#include "G4PVPlacement.hh"
#include "G4SystemOfUnits.hh"
#include "G4VPhysicalVolume.hh"
#include "G4LogicalVolume.hh"

class MyDetectorConstruction : public G4VUserDetectorConstruction {
public:
  MyDetectorConstruction() {}
  ~MyDetectorConstruction() override = default;
  G4VPhysicalVolume* Construct();

  void setSensitive();

private:
  G4LogicalVolume   *logSens;
};
```

Sensitive Detector



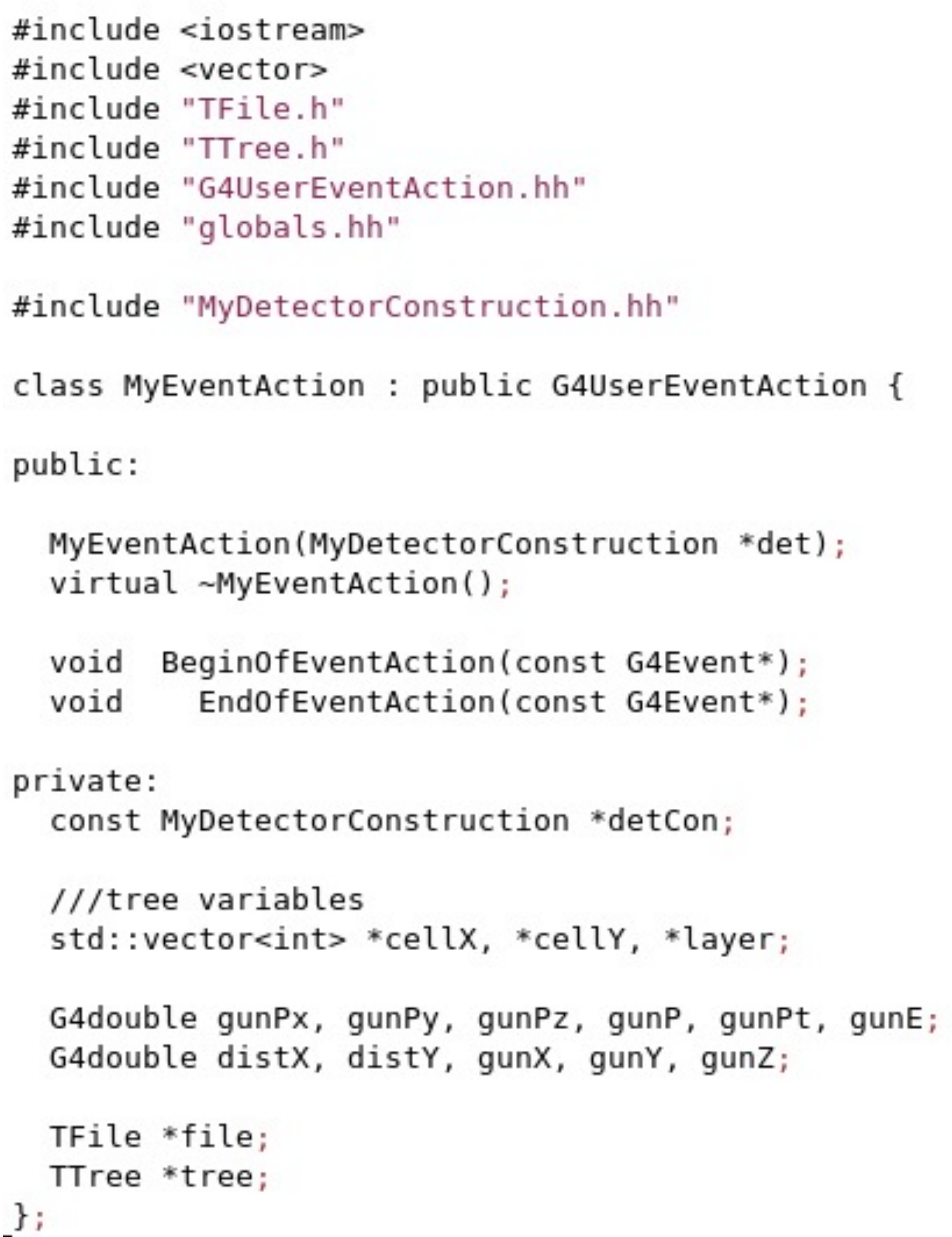
```
#include "G4VSensitiveDetector.hh"
#include "MyHit.hh"

class MyDetectorConstruction;
class G4Step;
class G4HCofThisEvent;
class G4TouchableHistory;

class MySensitiveDetector : public G4VSensitiveDetector {
public:
  MySensitiveDetector(MyDetectorConstruction*, G4String);
  ~MySensitiveDetector();

  void     Initialize(G4HCofThisEvent*HCE);
  G4bool   ProcessHits(G4Step*aStep, G4TouchableHistory*ROhist);
  void     EndOfEvent(G4HCofThisEvent*HCE);
  void     clear();
  void     DrawAll();
  void     PrintAll();

private:
  const MyDetectorConstruction *detector_;
  MyHitsCollection *calCollection_;
  G4int hcID_;

};
```

Primary Generator Action



```
#include "G4VUserPrimaryGeneratorAction.hh"
#include "globals.hh"
#include "G4ParticleGun.hh"
#include "G4ThreeVector.hh"

class G4Event;

class PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction {

public:
  PrimaryGeneratorAction();
  virtual ~PrimaryGeneratorAction();

  void GeneratePrimaries(G4Event*);
  G4ParticleGun* GetParticleGun() {return particleGun;};
private:
  G4ParticleGun*                particleGun;      //pointer a to G4  class
  G4double                      xVertex, yVertex, zVertex;
};
```

EventAction

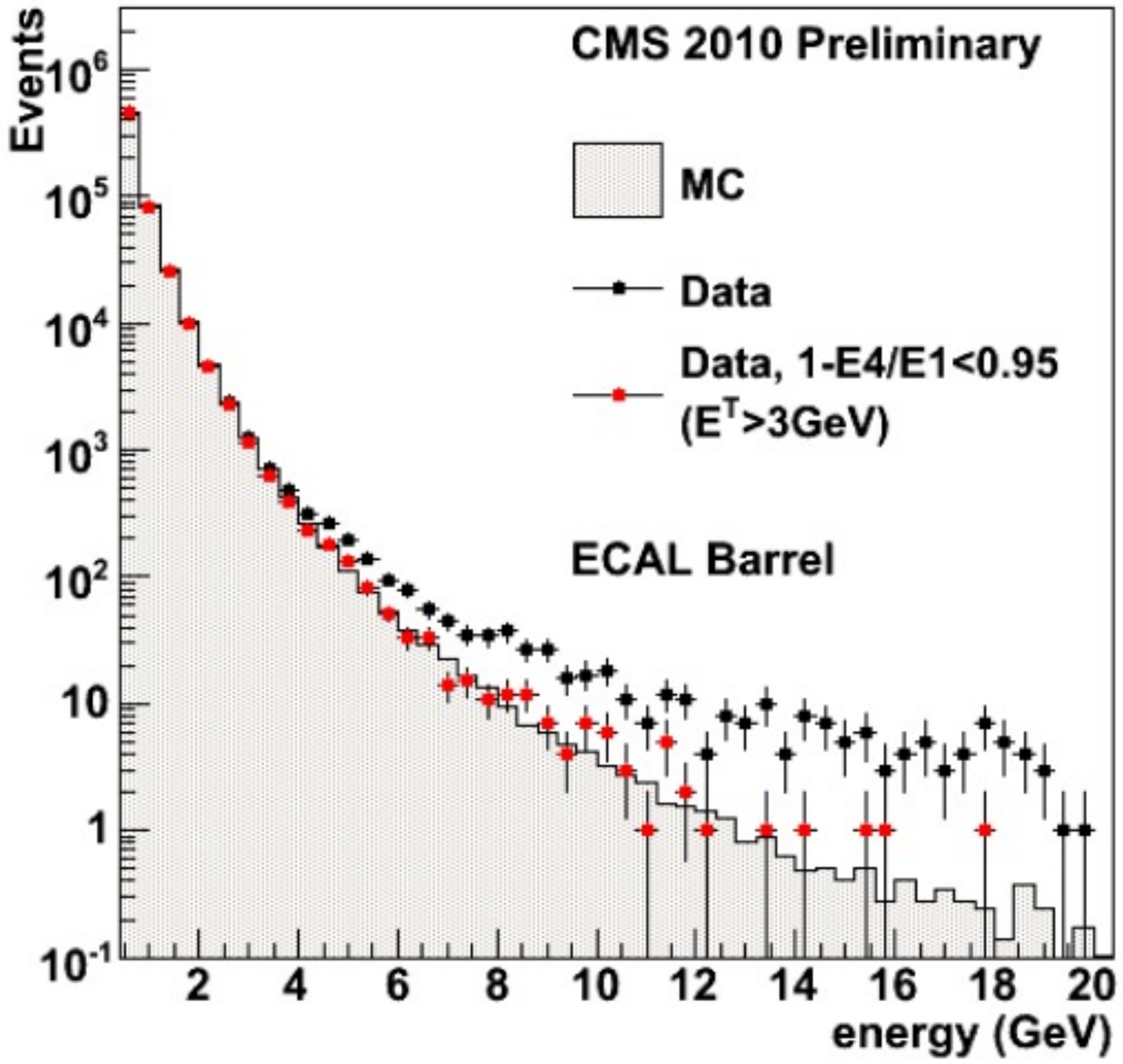


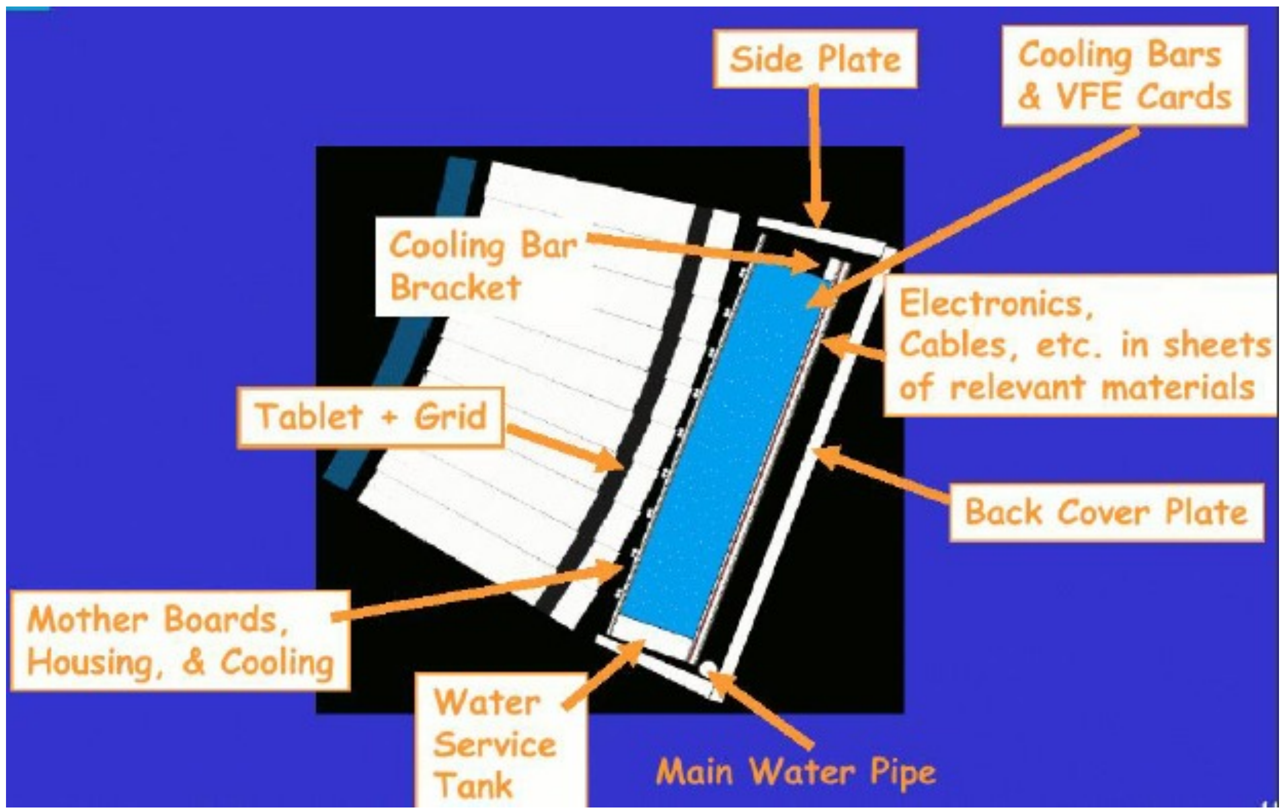
```
#include <iostream>
#include <vector>
#include "TFile.h"
#include "TTree.h"
#include "G4UserEventAction.hh"
#include "globals.hh"

#include "MyDetectorConstruction.hh"

class MyEventAction : public G4UserEventAction {

public:

  MyEventAction(MyDetectorConstruction *det);
  virtual ~MyEventAction();

  void  BeginOfEventAction(const G4Event*);
  void    EndOfEventAction(const G4Event*);

private:
  const MyDetectorConstruction *detCon;

  ///tree variables
  std::vector<int> *cellX, *cellY, *layer;

  G4double gunPx, gunPy, gunPz, gunP, gunPt, gunE;
  G4double distX, distY, gunX, gunY, gunZ;

  TFile *file;
  TTree *tree;
};
```

Additional Slides

Understanding Issues

- During early studies of collisions at the large hadron collider, CMS experiment observed some unusual number of events in their detector system

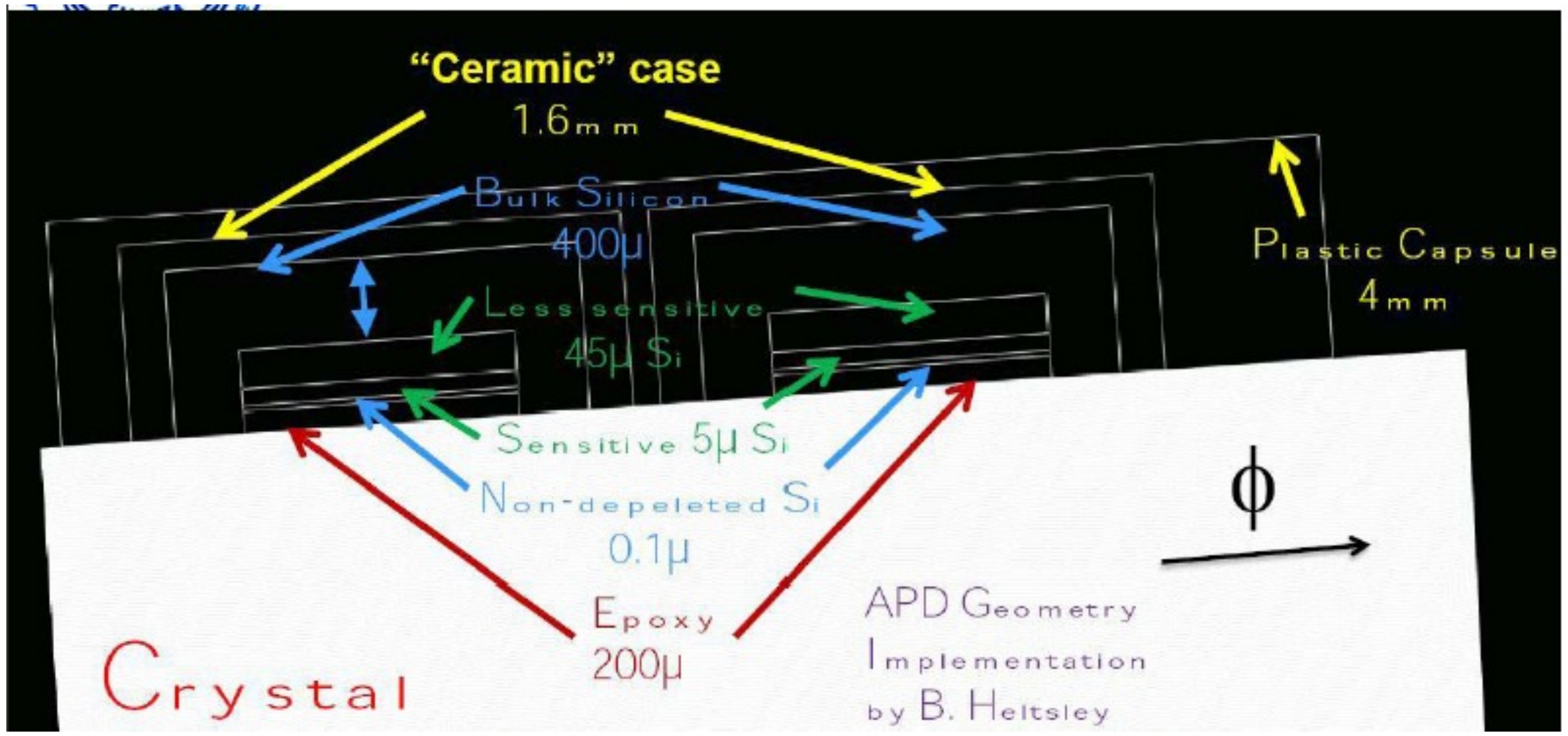


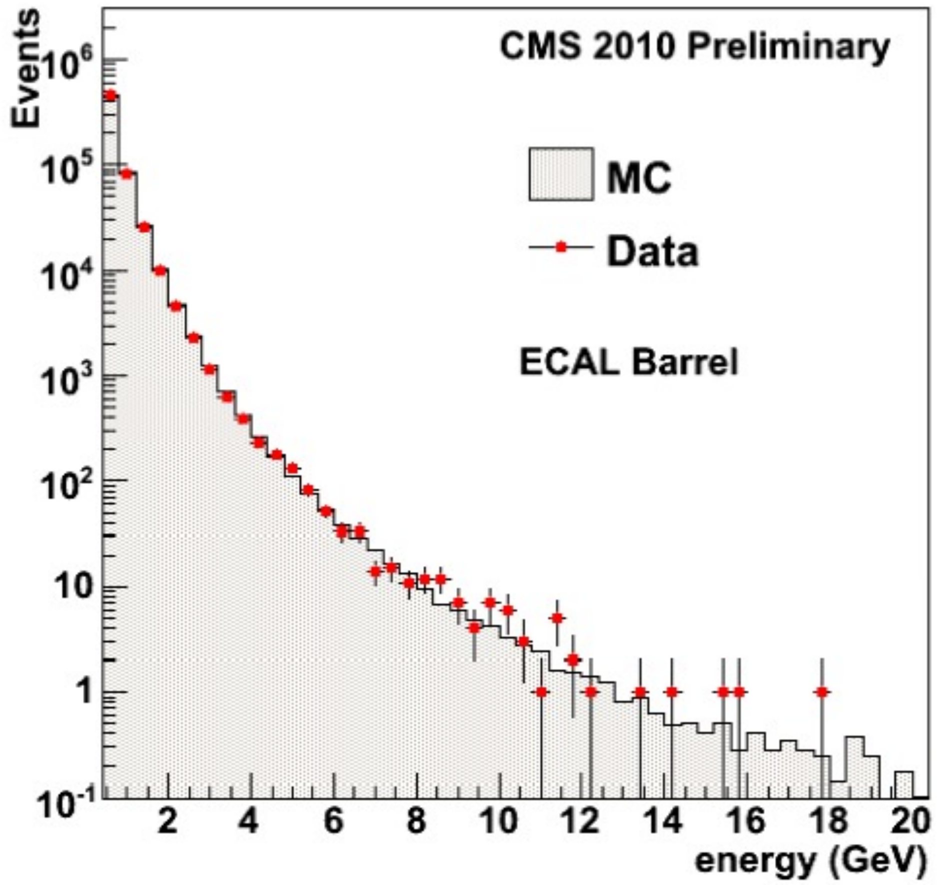


- CMS measured energy deposited in a crystal calorimeter. The excess was observed at higher energies

Understanding Issues

- CMS tried to understand the readout unit of the calorimeter and realised there could be excess energy if some heavily ionising particles go through the readout unit. This excess is not light produced in the crystal but charge deposited in the readout unit





- Simulating the readout unit removed the observed excess

- A qualitative as well as quantitative understanding of everything observed is crucial in providing results with the highest precision and confidence

- Simulation and Geant4 provide this confidence

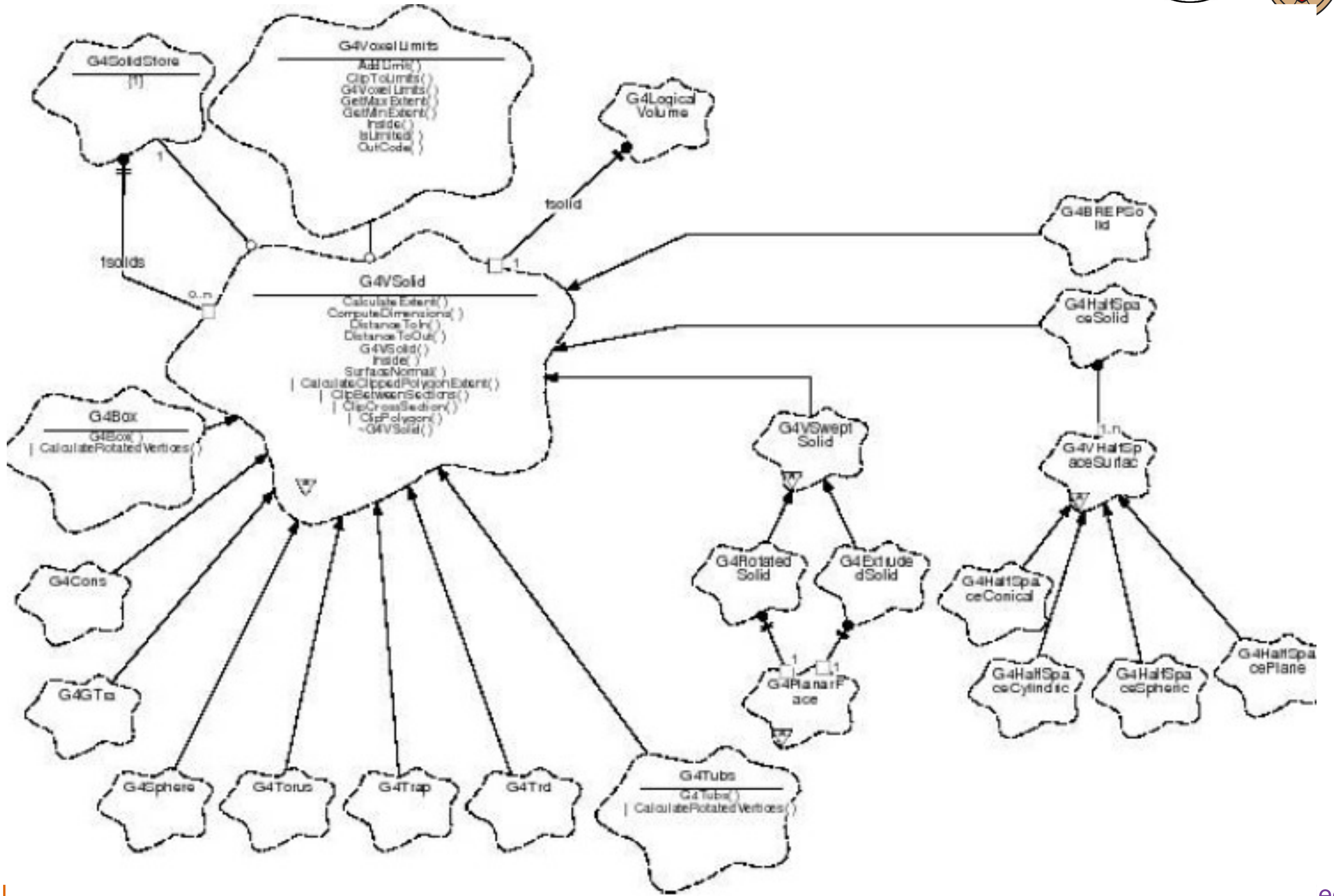
Bookkeeping Issues

- Connection from G4PrimaryParticle to G4Track
  G4int G4PrimaryParticle::GetTrackID()
  - Returns the track ID if this primary particle had been converted into G4Track, otherwise -1

- Connection from G4Track to G4PrimaryParticle
  G4PrimaryParticle* G4DynamicParticle::GetPrimaryParticle()
  - Returns the pointer of G4PrimaryParticle object if this track was defined as a primary or pre-assigned decay product, otherwise null

- G4VUserPrimaryVertexInformation, G4VUserPrimaryParticleInformation and G4VUserTrackInformation may be used for storing additional information
  - Information in G4VUserTrackInformation should be then copied to the user-defined trajectory class so that such information is kept until the end of the event

Solids



G4SolidStore
[1]
G4VoxelLimits
AddLimit()
ClipToLimits()
G4VoxelLimits()
GetMaxExtent()
GetMinExtent()
Inside()
IsLimited()
OutCode()
G4LogicalVolume
fsolid
fsolids
0..n
1
G4VSolid
CalculateExtent()
ComputeDimensions()
DistanceToIn()
DistanceToOut()
G4VSolid()
Inside()
SurfaceNormal()
| CalculateClippedPolygonExtent()
| ClipBetweenSections()
| ClipCrossSection()
| ClipPolygon()
~G4VSolid()
G4BREPSolid
G4HalfSpaceSolid
G4VSweptSolid
1..n
G4VHalfSpaceSurfac
G4Box
G4Box()
| CalculateRotatedVertices()
G4Cons
G4GTrd
G4Sphere
G4Torus
G4Trap
G4Trd
G4Tubs
G4Tubs()
| CalculateRotatedVertices()
G4RotatedSolid
G4ExtrudedSolid
G4PlanarFace
G4HalfSpaceConical
G4HalfSpaceCylindric
G4HalfSpaceSpheric
G4HalfSpacePlane

Hit Collection, Hit Map

- G4VHitsCollection is the common abstract base class of both G4THitsCollection and G4THitsMap

- G4THitsCollection is a template vector class to store pointers of objects of one concrete hit class type
  - A hit class (deliverable of G4VHit abstract base class) should have its own identifier (e.g. cell ID)

  - G4THitsCollection requires the user to implement their own hit class

- G4THitsMap is a template map class that stores keys (typically cell ID, i.e. copy number of the volume) with pointers of objects of one type

  - Objects may not be those of the hit class
    - All of the currently provided scorer classes use G4THitsMap with simple double

  - Since G4THitsMap is a template, it can be used by the sensitive detector class to store hits

Alternative Cross Sections

- Cross-section databases are available for low-energy neutrons
  - G4NDL available among the Geant4 distribution files
  - Livermore database (LEND) is also available
  - these are available with or without thermal cross-sections

- Cross section table is available for medium energy neutrons and protons
  - 14 MeV < E < 20 MeV

- Several alternatives exist for ion-nucleus cross-section
  - these are empirical and parametrised cross-section formulae with some theoretical insight
  - these are good for E/A < 10 GeV

- Alternative cross-sections also exist for pion cross-section

Data Driven Hadronic Models

- These are characterised by lots of data on
  - cross sections
  - angular distributions
  - multiplicities, etc.

- To get interaction length and final state, these models depend on interpolation of data
  - cross sections, Legendre coefficients, ..

- Examples:
  - neutrons with E < 20 MeV
  - coherent elastic scattering (pp, np, nn)
  - radioactive decays

Theory Driven Hadronic Models

- These are dominated by theoretical arguments (QCD, Glauber theory, exciton model, …)
- Final states (number and type of particles in the final sate with their energy and angular distributions) are determined by sampling theoretically calculated distributions
- This type of models is preferred as they are the most predictive
- Examples:
  - quark-gluon string models (projectiles with $E > 20$ GeV)
  - intra-nuclear cascade models (intermediate energies)
  - nuclear de-excitation and break-up

Parametrised Hadronic Models

- Current versions do not contain any parametrised version. In versions preceding Geant4 10.0, two models existed. They were re-engineered versions of the Fortran Gheisha code used in Geant3

- These models depended mostly on fits to data with some theoretical guidance

- Two such models existed:
  - Low Energy Parametrised (LEP) for $E < 20$ GeV
  - High Energy Parametrised (HEP) for $E > 20$ GeV
  - each type referred to a collection of models (one for each type of hadron)

- These codes were fast and existed for all types of particles. But they were not detailed enough and there was no-one to maintain these codes