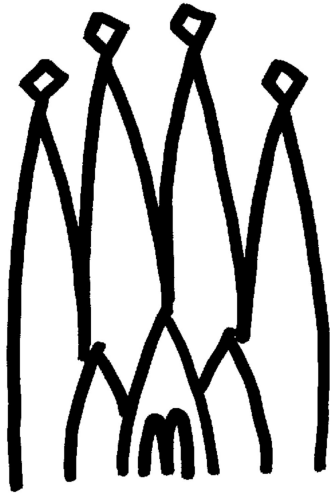
SPDGeoModel and Sampo development introduction

Part III
Gaudi + Sampo

Gaudi: overview

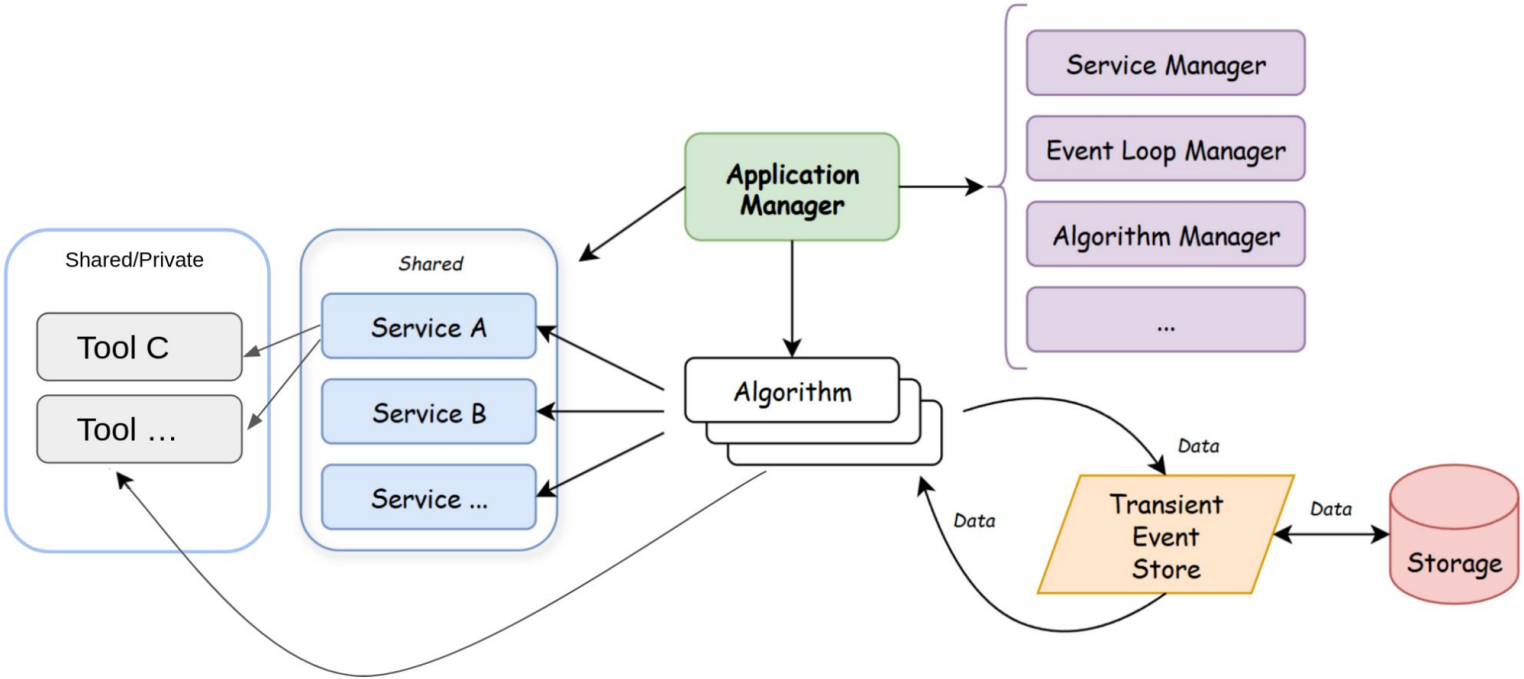


- C++ HEP framework
- 25+ years of development, originally from LHCb
- Key idea: algorithms manipulate data
- Multithreading in GaudiHive
- Configurable components
- Configuration via python

Useful links:

- Gaudi docs (very outdated, but anyway): https://gaudi.web.cern.ch/gaudi/old/GDG.html
- Gaudi repo: https://gitlab.cern.ch/gaudi/Gaudi
- Gaudi tutorial: https://git.jinr.ru/lsimbir/gauditutorial

Gaudi: architecture (single-threaded)



Gaudi: main components

DataObjects - any quantity-like objects (hits, tracks, etc.)

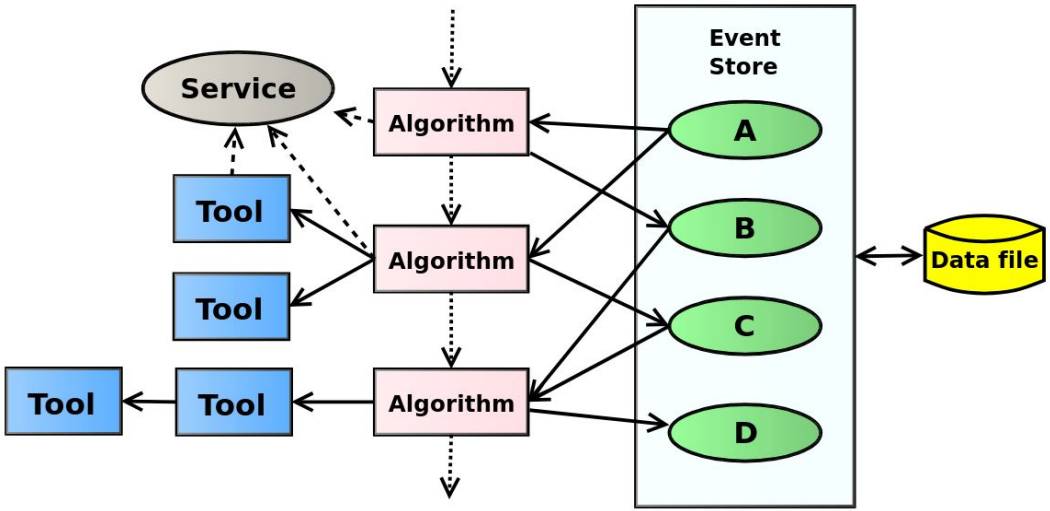
Algorithms - anything that can be represented as procedure

Services - global objects used by other components

Tools - encapsulated subprocedures that can be injected to other components

Transient event store (TES) - service which stores data objects and thus allows components to exchange data

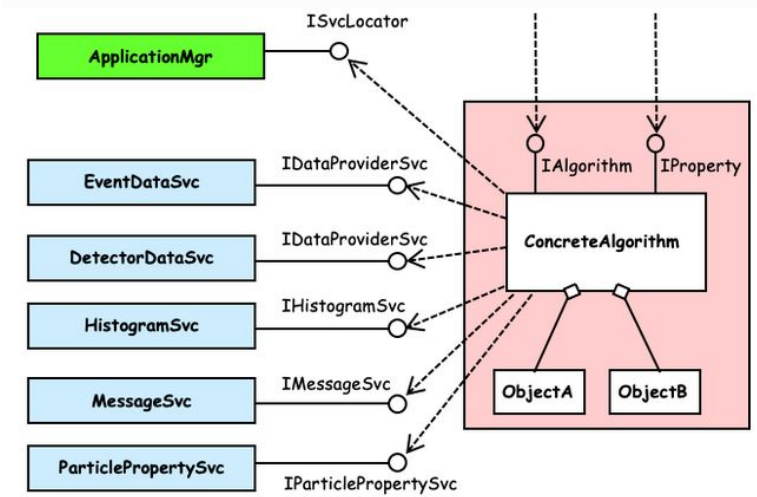
Example of control (dashed lines) and data (solid lines) flow in Gaudi



Gaudi: interfaces

In programming, an **interface** serves as a contract that defines a set of methods or properties that a class must implement. In C++ interfaces are implemented as abstract classes.

- in Gaudi all components follow at least one interface
- in Gaudi components are addressed through their interfaces
- interface classes are prefixed with “I”: e.g. IMagneticFieldSvc



The picture is outdated, but illustrates the concept very well

Gaudi: Algorithms

All algorithms follow IAlgorithm interface and must be derived from Algorithm base class, also base class constructor must be used.

```
1   #pragma once
2
3   #include "GaudiKernel/Algorithm.h"
4
5   class ExampleAlg : public Algorithm{
6
7   public:
8       using Algorithm::Algorithm;
9
10      StatusCode initialize() override;
11      StatusCode execute() override;
12      StatusCode finalize() override;
13  };
```

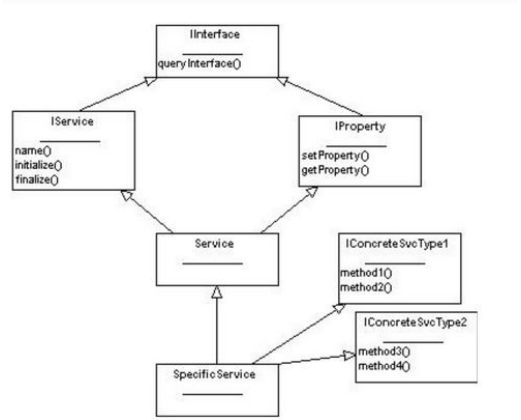
initialize(): called when algorithm is created. Initializing resources, creating histograms…

execute(): called once for every event. This is where most of the real action should take place

finalize(): called after event loop finishes. Closing files, summarizing results…

Gaudi: Services

Services must implement custom interfaces.
Services are called only by request, not once per event as it was for algorithms



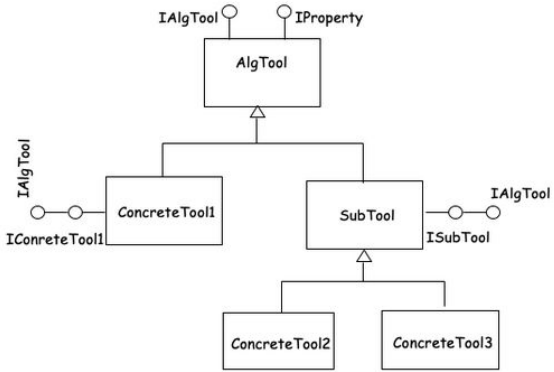



```cpp
#pragma once

#include "GaudiKernel/IInterface.h"

class IExampleSvc : virtual public IInterface {
public:
    // InterfaceID
    DeclareInterfaceID(IExampleSvc, 1, 0);

    virtual void callSvc() = 0;
};
```

```cpp
#pragma once

#include <string>

#include "IExampleSvc.h"
#include "GaudiKernel/Service.h"

class ExampleSvc : public extends<Service, IExampleSvc> {
public:

    // using base class constructor
    using extends::extends;

    StatusCode initialize() override;
    StatusCode finalize() override;

    void callSvc() override;

private:
    Gaudi::Property<std::string> m_stringProperty {this, "stringProperty", "default"};
};
```

Gaudi: Tools

Encapsulated piece of code that can be injected to any other component. They may serve as “light” algorithms.

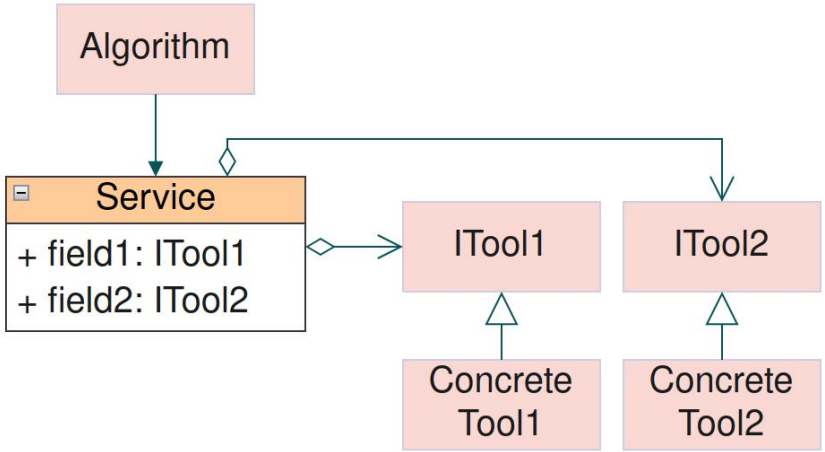



```cpp
#pragma once
#include "GaudiKernel/IAlgTool.h"

class IExampleToolFirst : virtual public IAlgTool {
public:
  /// InterfaceID
  DeclareInterfaceID(IExampleToolFirst, 1, 0);

  virtual void doThis() const = 0;
};


class IExampleToolSecond : virtual public IAlgTool {
public:
  /// InterfaceID
  DeclareInterfaceID(IExampleToolSecond, 1, 0);

  virtual void doThat() const = 0;
};
```

```cpp
#pragma once

#include "GaudiKernel/AlgTool.h"
#include "IExampleTool.h"

class ExampleTool : public extends<AlgTool, IExampleToolFirst, IExampleToolSecond> {
public:
  /// Standard Constructor
  using extends::extends;

  /// IExampleToolFirst interface
  void doThis() const override;

  // IExampleToolSecond interface
  void doThat() const override;

  StatusCode initialize() override;
  StatusCode finalize() override;

private:
  /// Properties
  Gaudi::Property<std::string> m_stringProperty{ this, "stringProperty", "default" };
};
```

Gaudi: Tools and Services

Concept: Services encapsulate high-level logic whereas Tools can be used for details implementation. Concrete Tools can be selected via job options.



Gaudi: TES and DataObjects

- TES is designed to operate with DataObjects
- There are read and write handles to simplify access to TES

```
17 class HepMCReaderAlg : public Algorithm{
18 public:
19   // Base class ctor
20   using Algorithm::Algorithm;
21
22   StatusCode initialize() override;
23   StatusCode execute() override;
24   StatusCode finalize() override;
25
26 private:
27   std::shared_ptr<HepMC3::Reader> m_reader;
28
29   DataObjectWriteHandle<std::shared_ptr<HepMC3::GenEvent>> m_hepmcWriteHandle{this, "hepmc", TDSAddresses::MC::HepMC};
30   DataObjectWriteHandle<long> m_seedWriteHandle{this, "seed", TDSAddresses::MC::Seed};
31
32   Gaudi::Property<std::string> m_filePath{this, "filePath", "", "Input HepMC file path"};
33 };
```

```
21 StatusCode HepMCReaderAlg::execute(){
22   static long nEvt = 0;
23
24   auto hepmcevt = std::make_shared<HepMC3::GenEvent>(HepMC3::Units::GEV, HepMC3::Units::MM);
25   if(!m_reader->read_event(*hepmcevt)){
26     error() << "Can't read HepMC event" << endmsg;
27     return StatusCode::FAILURE;
28   }
29
30   m_hepmcWriteHandle.put(std::move(hepmcevt));
```

Gaudi: Python configuration scripts (job options)

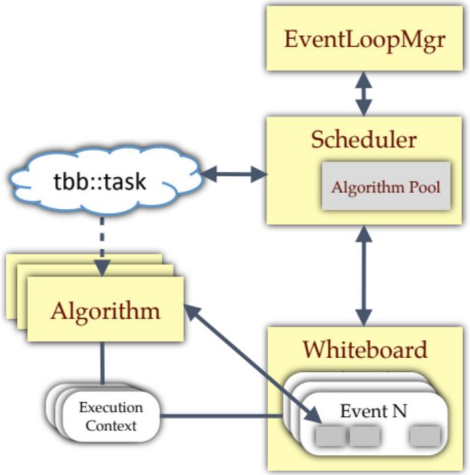
- *gaudirun.py* is for processing job options and launching the job
- *run* is autogenerated script for setting up the environment

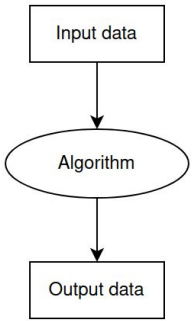
```python
# config.py

from Configurables import ExampleAlg, ExampleTool

from Gaudi.Configuration import ApplicationMgr

evt_max = 1
evt_sel = "NONE"

my_algo = ExampleAlg("A1")
my_algo.stringProperty = "Hello!"

tool = ExampleTool("ExampleTool1")
tool.stringProperty = "I am private tool"
my_algo.PrivateTool = tool

tool2 = ExampleTool("ExampleTool2")
tool2.stringProperty = "I am public tool"
my_algo.PublicTool = tool2

# create ApplicationMgr and start Gaudi app (C++)
ApplicationMgr(
    EvtMax=evt_max,
    EvtSel="NONE",
    TopAlg=[my_algo]
)
```

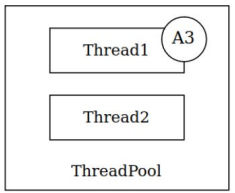
./run gaudirun.py path/to/options.py

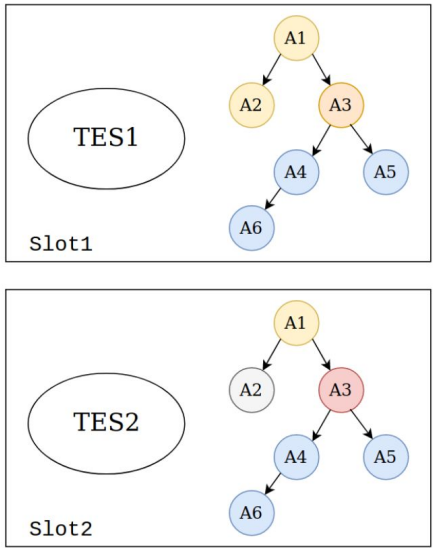
GaudiHive: architecture

Concept: Many events are processed simultaneously. Scheduler analyzes event execution-state graphs, finds algorithm that can be launched, picks it from AlgPool, packs it with its input data into a task and sends to the task queue.

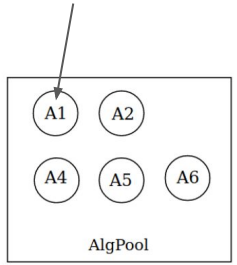








Can be cloned or re-entrant

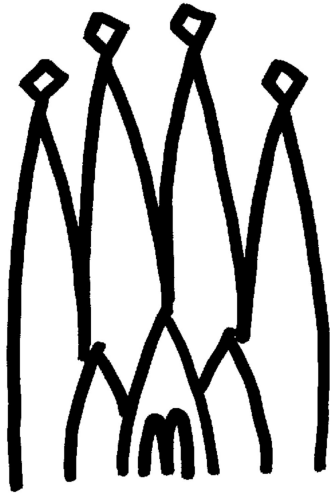


Sampo: overview

Gaudi-based offline SPD software (one day it will be!!!)

Current functionality:

- HepMC3 as main format for generators
- Pythia8 for events generation (MT)
- GeoModel for detector description
- Geant4 for simulation (MT)
- Magnetic field description features



Gaudi logo. I hope one day we will have our own.

Useful links:

- Sampo base Docker image for development: https://git.jinr.ru/spd/spd-sw/bisgm
- Sampo repo: https://git.jinr.ru/spd/spd-sw/sampo/ (develop branch)

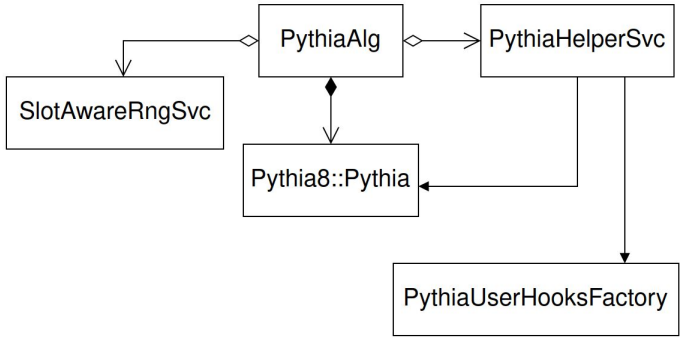
Sampo: for developers

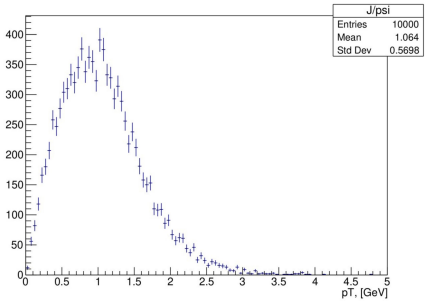
- use Sampo base Docker image for development: https://git.jinr.ru/spd/spd-sw/bisgm

- read through CONTRIBUTING.md to follow our git flow
- apply clang-format to your changes before MR

Pythia8

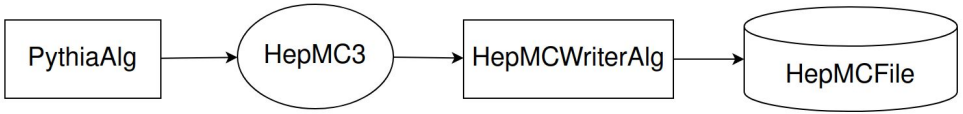
Concept: PythiaHelperSvc creates Pythia8 generator instances, equips them with random number generators and applies all requested user hooks. Generation is performed in PythiaAlg.



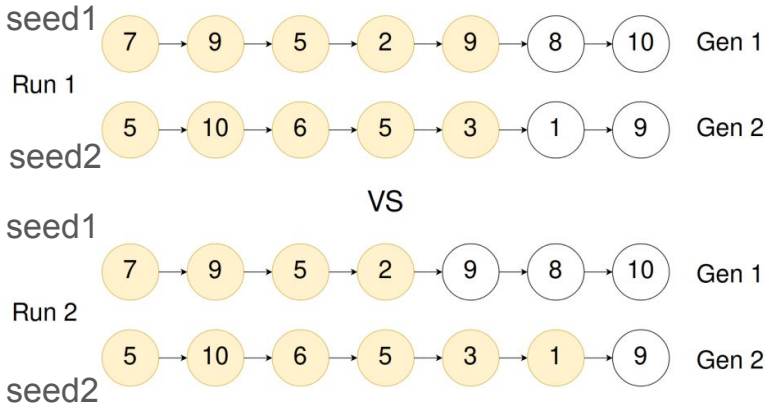


pp, √s = 27 GeV

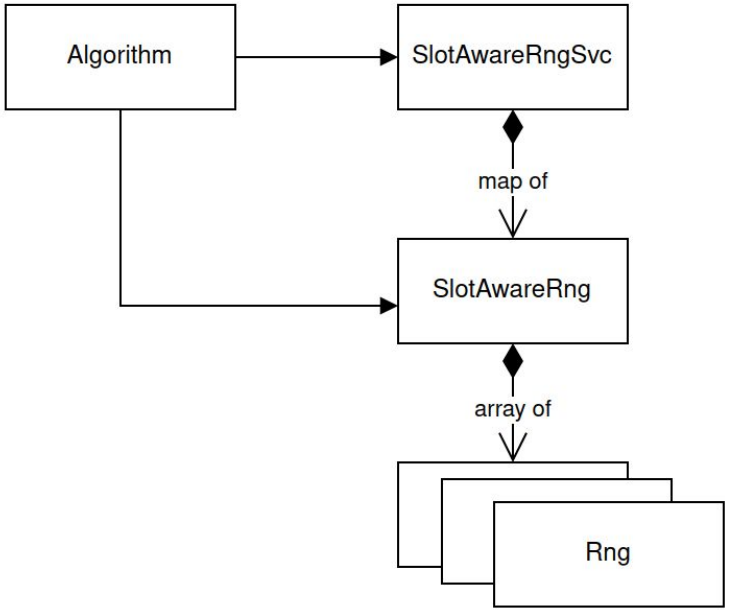
```
1   from Configurables import PythiaAlg, PythiaHelperSvc, HepMCWriterAlg
2   from Gaudi.Configuration import *
3
4   pythiaHelper = PythiaHelperSvc()
5   pythiaHelper.settingsFileName = "/workspaces/sampo/Testing/Pythia8/data/testconf.cmnd"
6
7   hepmcWriter = HepMCWriterAlg()
8   hepmcWriter.outputFormat = "Ascii"
9   hepmcWriter.filePath = "testPythia.dat"
10
11  EventDataSvc(ForceLeaves=True)
12
13  # create ApplicationMgr and start Gaudi app (C++)
14  ApplicationMgr(
15      EvtMax=50000,
16      EvtSel="NONE",
17      TopAlg=[PythiaAlg(), hepmcWriter]
18  )
```



Problems introduced by MT



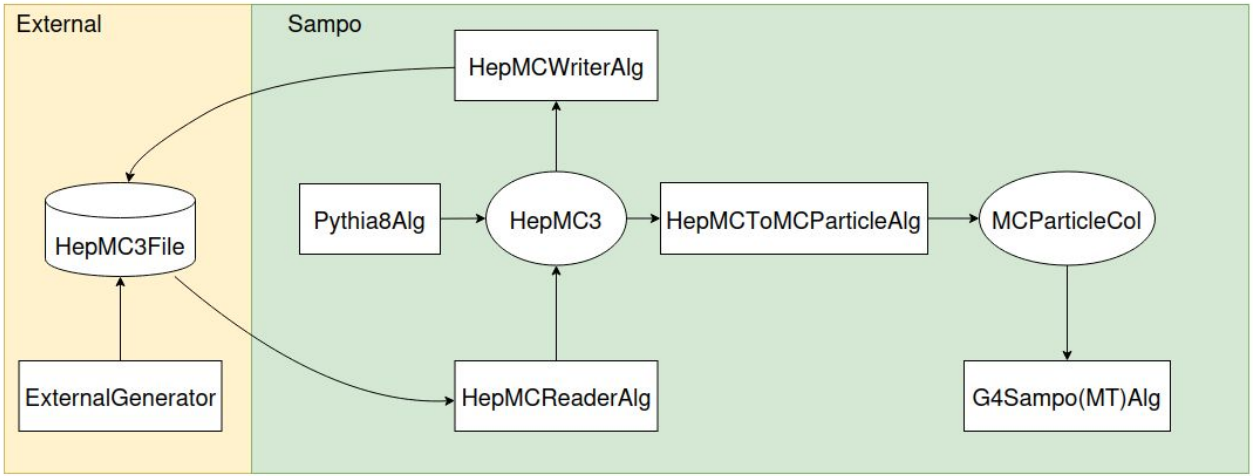
Each clone has its own random number generator, seeded only once at the beginning. Reproducibility is not guaranteed. Also algorithms can be re-entrant and thus need more than one random number generator.



64 bit xxhash as hf
seed = hf(init_seed)^hf(alg_name)^hf(evt_num).
Reproducibility is guaranteed.

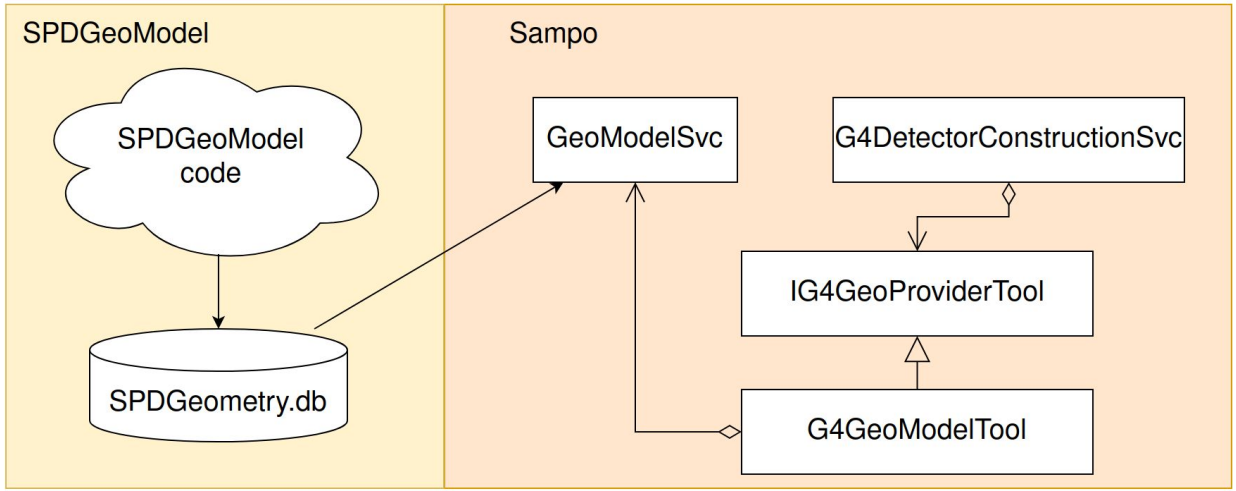
Sampo: HepMC3

- The HepMC package is an object oriented C++ event record for HEP MC generators and simulation
- We assume all generators must be able to convert their output to this format
- Allows to decouple generation and simulation processes



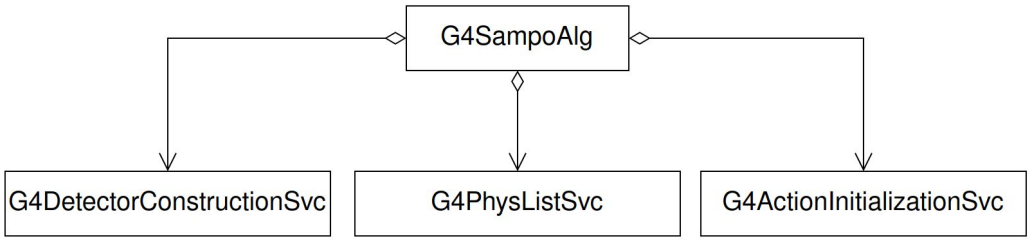
Sampo: GeoModel

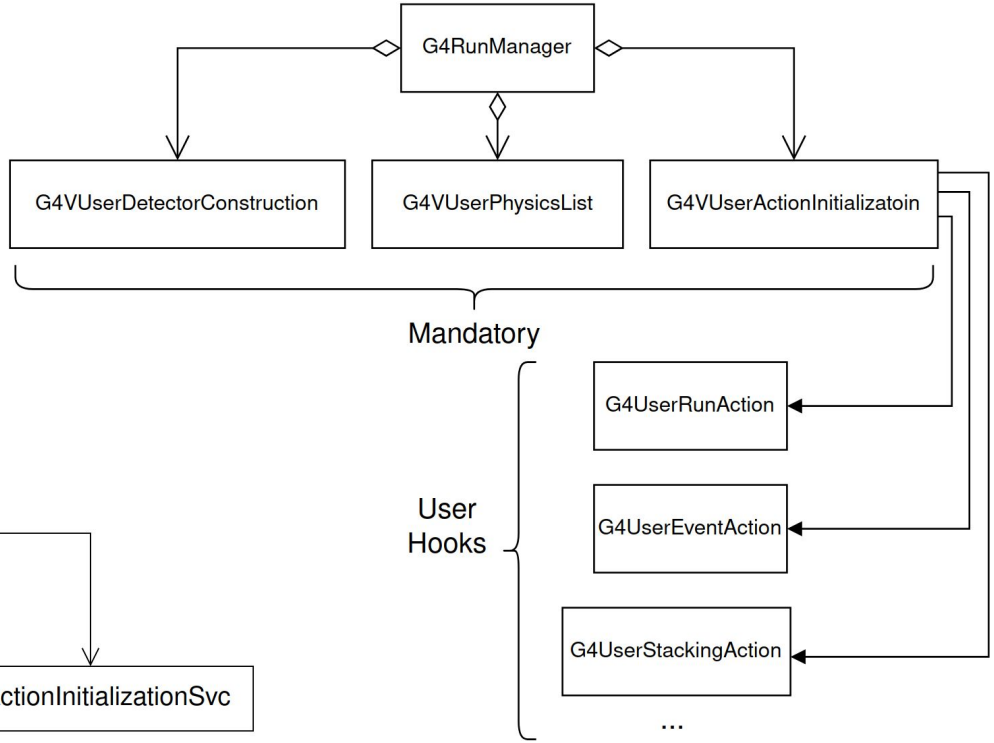
- SPDGeoModel project for SPD geometry creation
- GeoModelSvc reads geometry from DB



Sampo: Geant4

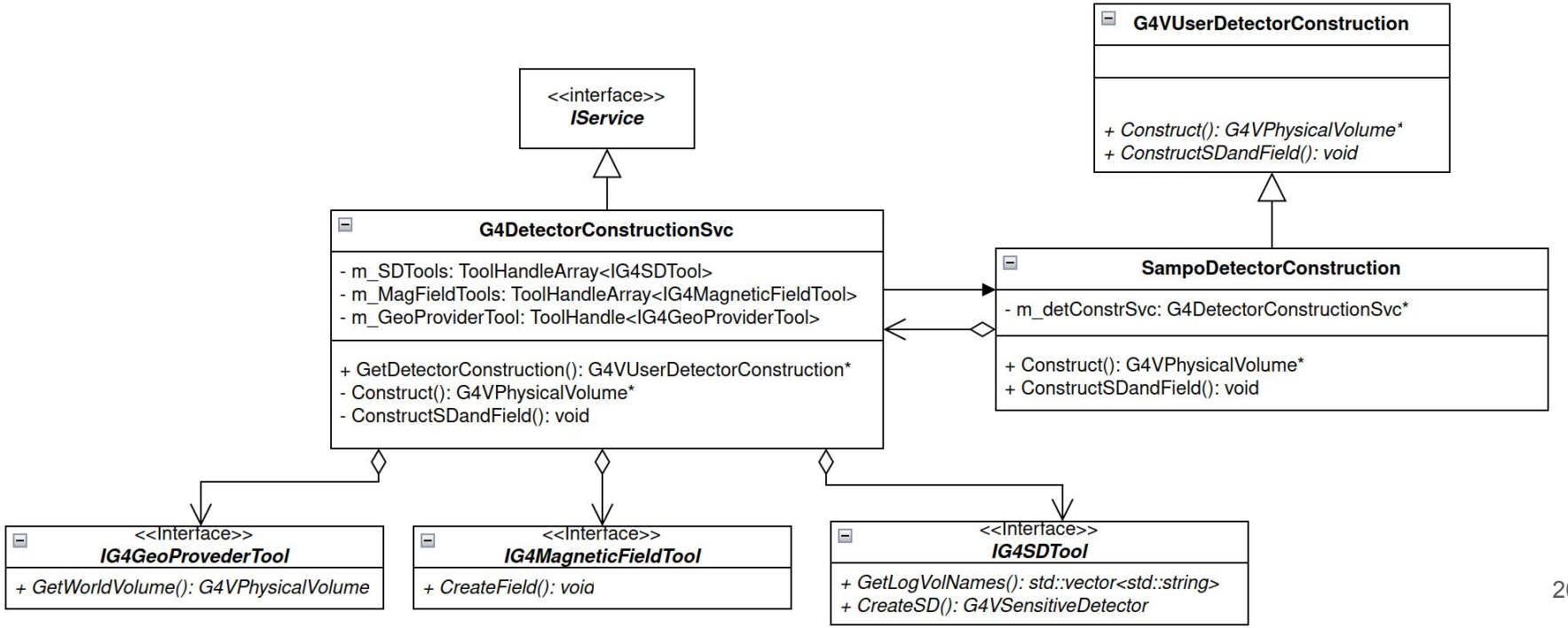




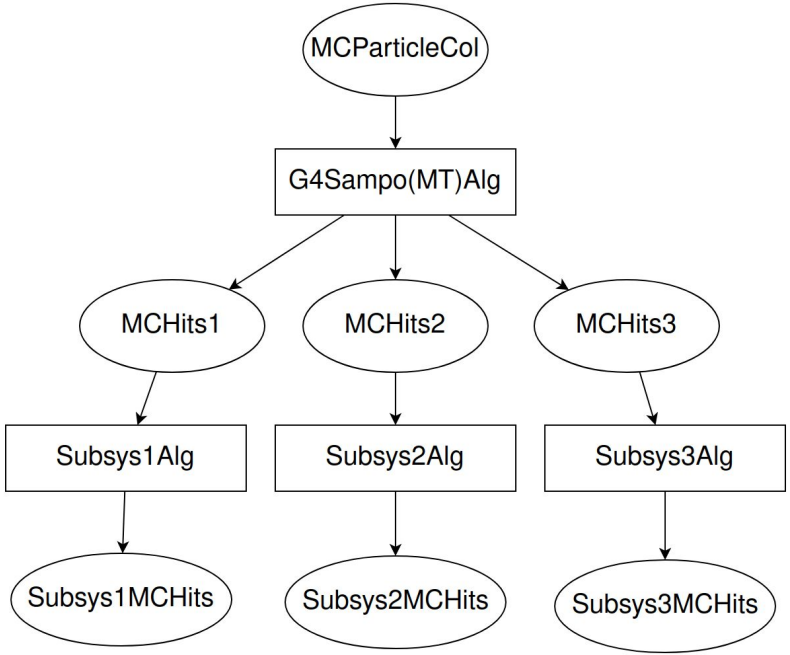


Sampo: Geant4

Concept: G4DetectorConstructionSvc wraps itself into adapter class SampoDetectorConstruction. Main work of detector and field creation is delegated to contained tools.



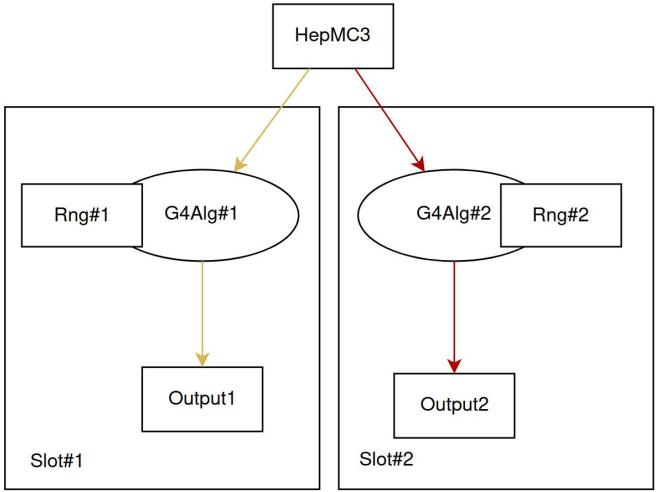
Sampo: Geant4

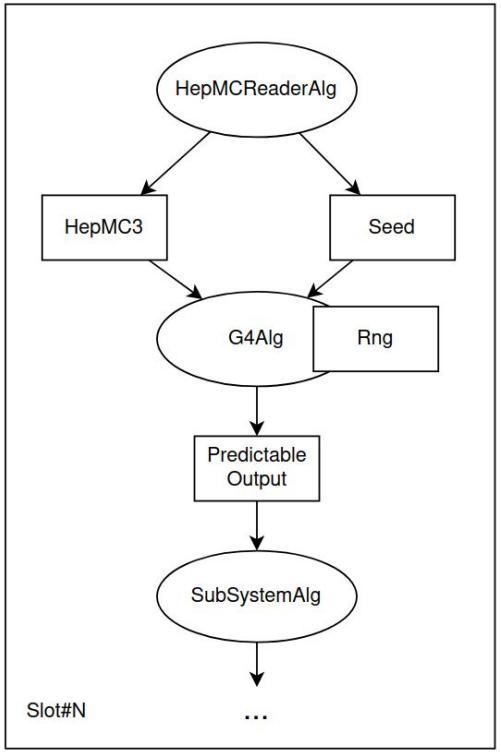


- G4Sampo(MT)Alg processes MCParticles from MCParticleCol
- G4ParticleGunAlg or HepMCToMCParticleAlg to create MCParticles
- G4Sampo(MT)Alg produces collections of universal MCHits for each subsystem
- Only one type of sensitive detector in Geant4

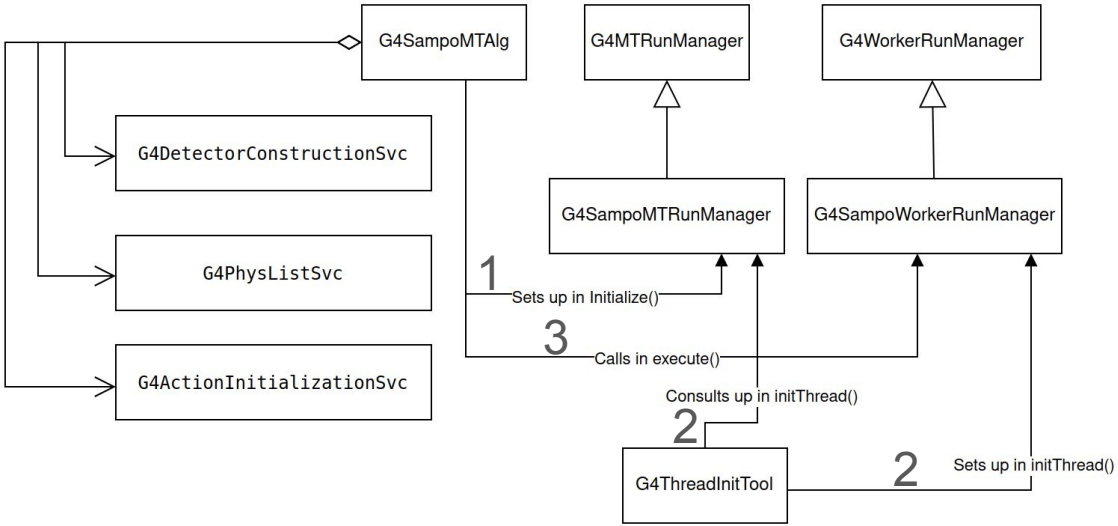
Problems arising from multithreading: reproducibility

We can’t predict by which of G4Alg instances this certain HepMC3 record will be processed. As long as random number generators in G4Algs are in different states the result is unpredictable.





Sampo: multithreaded Geant4



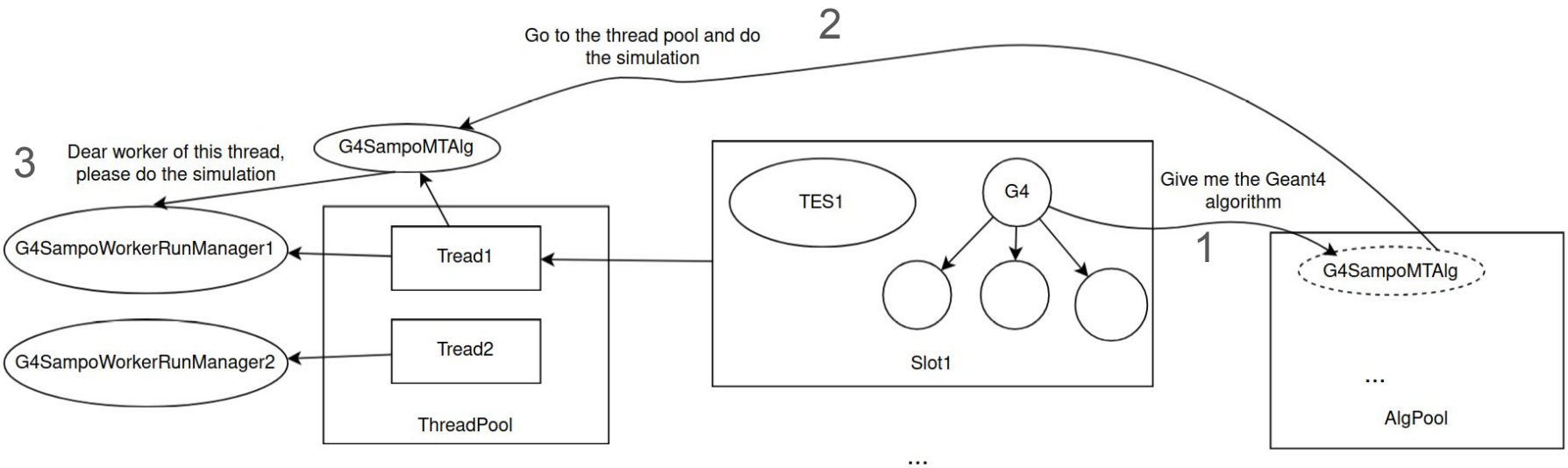
1 - First created instance of G4SampoMTAlg sets up a global G4SampoMTRunManager

2 - Whenever new thread is created it automatically gets equipped with its own instance of thread local G4SampoWorkerRunManager

3 - Whenever G4SampoMTAlg appears in one of the threads of the thread pool it acquires G4SampoWorkerRunManager of that thread and performs simulation

Advantages: simple
Disadvantages: fragile

Sampo: multithreaded Geant4



Sampo: magnetic field (thread unsafe so far)

Concept: Field stored as 3D map in SQLite DB. Trilinear interpolation for evaluation at target point.

