

# The GENIE framework

Costas Andreopoulos<sup>1,2</sup>

<sup>1</sup>University of Liverpool, <sup>2</sup>STFC Rutherford Appleton Laboratory

March 10, 2014



UNIVERSITY OF  
LIVERPOOL



Science & Technology Facilities Council  
Rutherford Appleton Laboratory

# Outline

- Gabe thought it might be useful to talk about the GENIE framework.
- Despite the fact that GENIE popularity has soared, nobody else had asked me to give such a talk in the past 5-6 yrs.
- I suspect this is probably good
  - A good framework is one that you do not need to talk about!
  - Works reliably, allows people to code-up physics and generate events without having to fight with it.
  - And typical GENIE users are largely shielded from GENIE internals. I suspect that most of you, as users, never had to look beyond one of the built-in user apps.

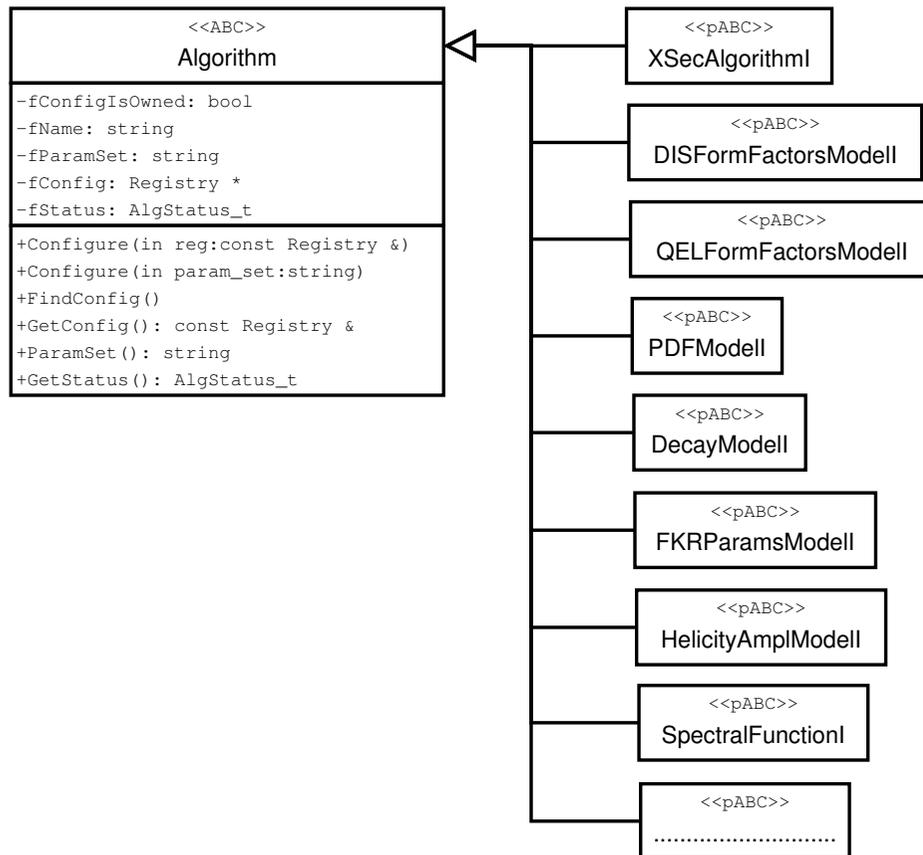
Yet another 9-hr flight to Chicago in a plane with an entertainment system from the 70s (thanks AA) allowed me to put together some slides, with the **basics that one needs to know to code-up new models in GENIE.** Can discuss this more throughout the week as we see specific examples.

The UML diagrams shown here were made many yrs ago and may be outdated. Do not use them as a reference for class method and data member names!

# History and Status

- GENIE at **2003**: O(5,000) lines of code  
A bunch of ROOT macros for cross-section calculations
- GENIE at **2007**: O(100,000) lines of code  
First official release, incl. complete C++ framework and physics proven to be fully equivalent to neugen 3.?, a fortran generator used in MINOS
- GENIE at **2014**: O(170,000) lines of code  
+ Physics improvements, numerous tools for parameter fitting, validation with data, experimental interfaces, ...
- Bulk of GENIE framework devel in 2004-2006. Little change since then.
- Very heavily influenced by MINOS work and discussions with Robert Hatcher, George Irwin, Brett Viren, Sue Kasahara, Nick West and others.
- I have changed some of my views on software in the past 10 yrs, I am somewhat less purist and my Design Patterns book is no longer on the top of the stack.
- Although I would have done some things differently, I think that the GENIE framework, albeit with a bit of clutter around it, is awesome and succesful.

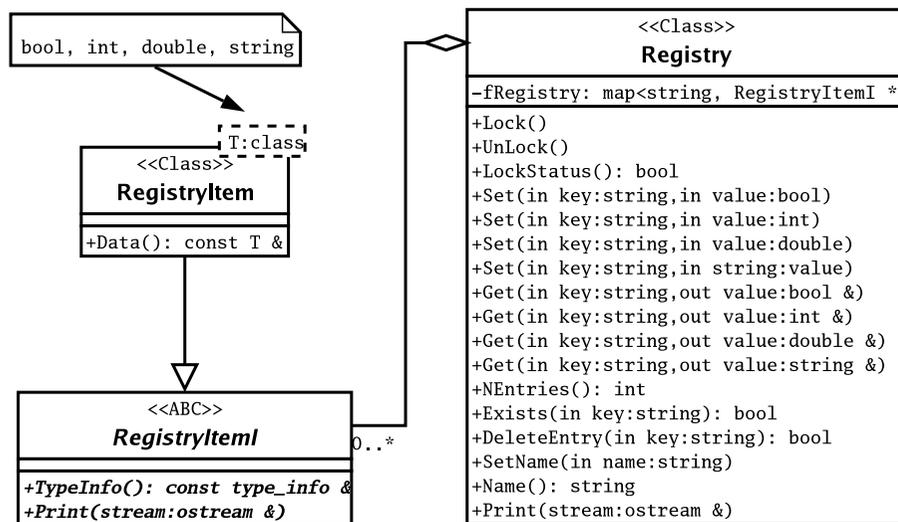
# Algorithms



- In GENIE, pretty much everything that does something to simulate an event has **Algorithm** at the root of its inheritance tree (is an algorithm).
- There are very different algorithms (particle decayers, differential cross-section models, numerical integrators) but they all have some things a common (a system to ID them, a system to ID their configuration, a system to access their configuration data etc). This is defined in the Algorithm base class.
- Then, we have a series of classes which defines the additional interface methods required for specific computations (eg `XSecAlgorithmI` defines the extra methods that need to be implemented by cross-section calculation code). Typically, you will be subclassing one of these interface classes and very rarely you would need to define new interfaces yourself.

# Algorithm configuration

Algorithm configurations are read from XML files and are held in named, type-safe "parameter" → "value" maps (Registries).



Notice that the "Default" configuration on the left specifies nothing. The default parameter values are obtained from **UserPhysicsOptions.xml**, the main configuration file that users interact with. In general, in algorithm-specific XML files, one defines configuration parameters only a) to provide non-default configuration or b) to set parameters we do not expose to the users (i.e. parameters not specified in UserPhysicsOptions.xml). Be cautious: Whatever you put there overrides defaults.

Example configurations for PYTHIA6 hadronization algorithm:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<alg_conf>
  <!--
  Default PYTHIA
  -->
  <param_set name="Default">
  </param_set>

  <!--
  Standard PYTHIA
  -->
  <param_set name="Standard">
    <param type="double" name="SSBarSuppression"      0.30 </param>
    <param type="double" name="GaussianPt2">        0.36 </param>
    <param type="double" name="NonGaussianPt2Tail">  0.01 </param>
    <param type="double" name="RemainingEnergyCutoff"> 0.80 </param>
  </param_set>

  <!--
  Tuned parameters as used by NUX (see A.Rubbia's talk @ NuINT01)
  -->
  <param_set name="NUX">
    <param type="double" name="SSBarSuppression"      0.21 </param>
    <param type="double" name="GaussianPt2">        0.44 </param>
    <param type="double" name="NonGaussianPt2Tail">  0.01 </param>
    <param type="double" name="RemainingEnergyCutoff"> 0.20 </param>
  </param_set>
</alg_conf>
  
```

# Nested algorithms

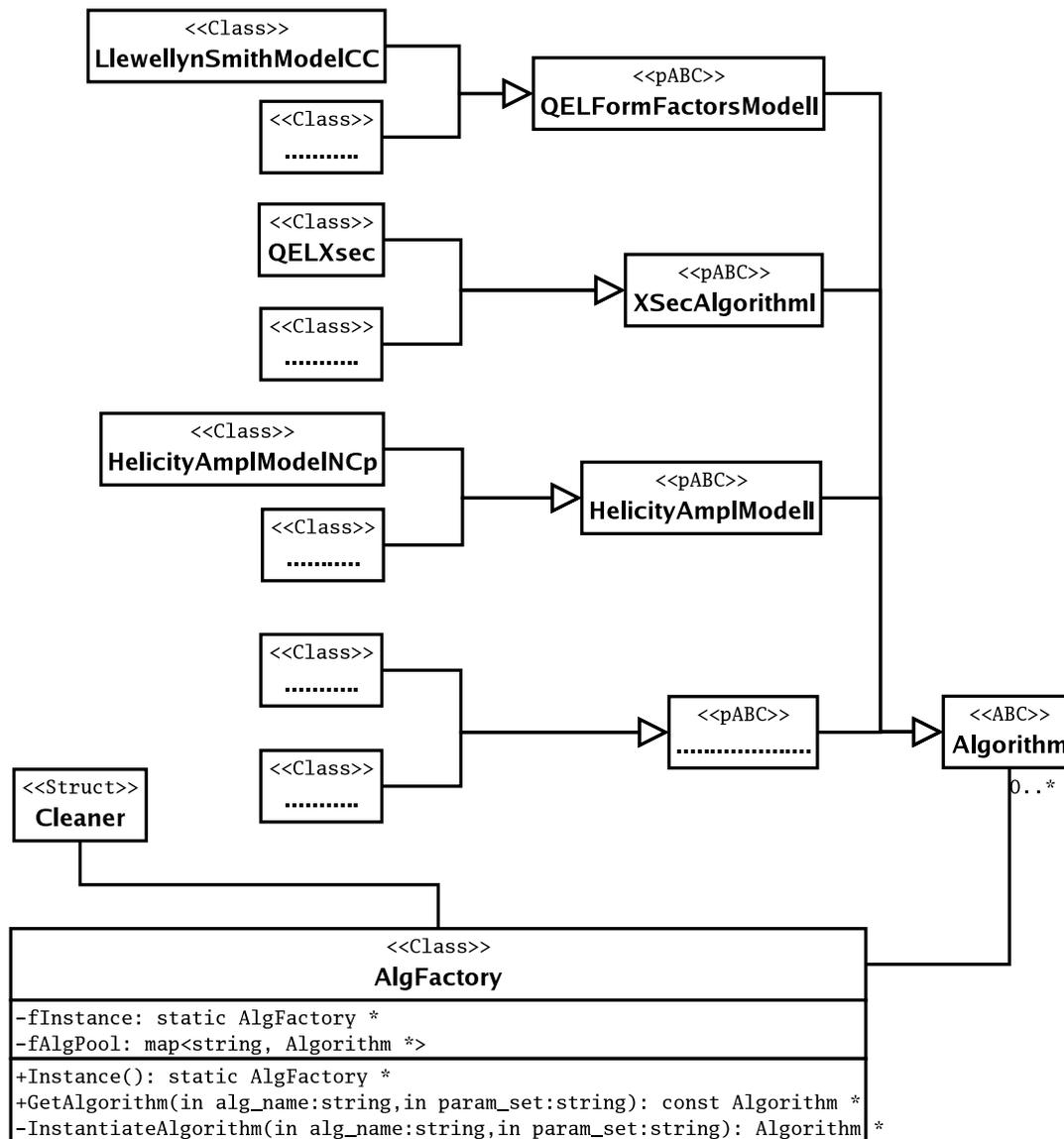
- Crucially, an algorithm configuration can specify other algorithms.

For example, the configuration for the DIS differential cross-section model specifies a **structure function** model, a **numerical algorithm** for cross-section integration and a **hadronization algorithm** to help obtain correction factors in the transition region:

```
<param type="alg" name="SFAlg" >  
  genie::BYStrucFunc/Default </param>  
<param type="alg" name="XSec-Integrator" >  
  genie::DISXSec/Default </param>  
<param type="alg" name="Hadronizer" >  
  genie::KNOHadronization/Default </param>
```

- One can build a complex call tree (specify who calls what) entirely using the configuration files.
- You can tweak the call tree without having to recompile.

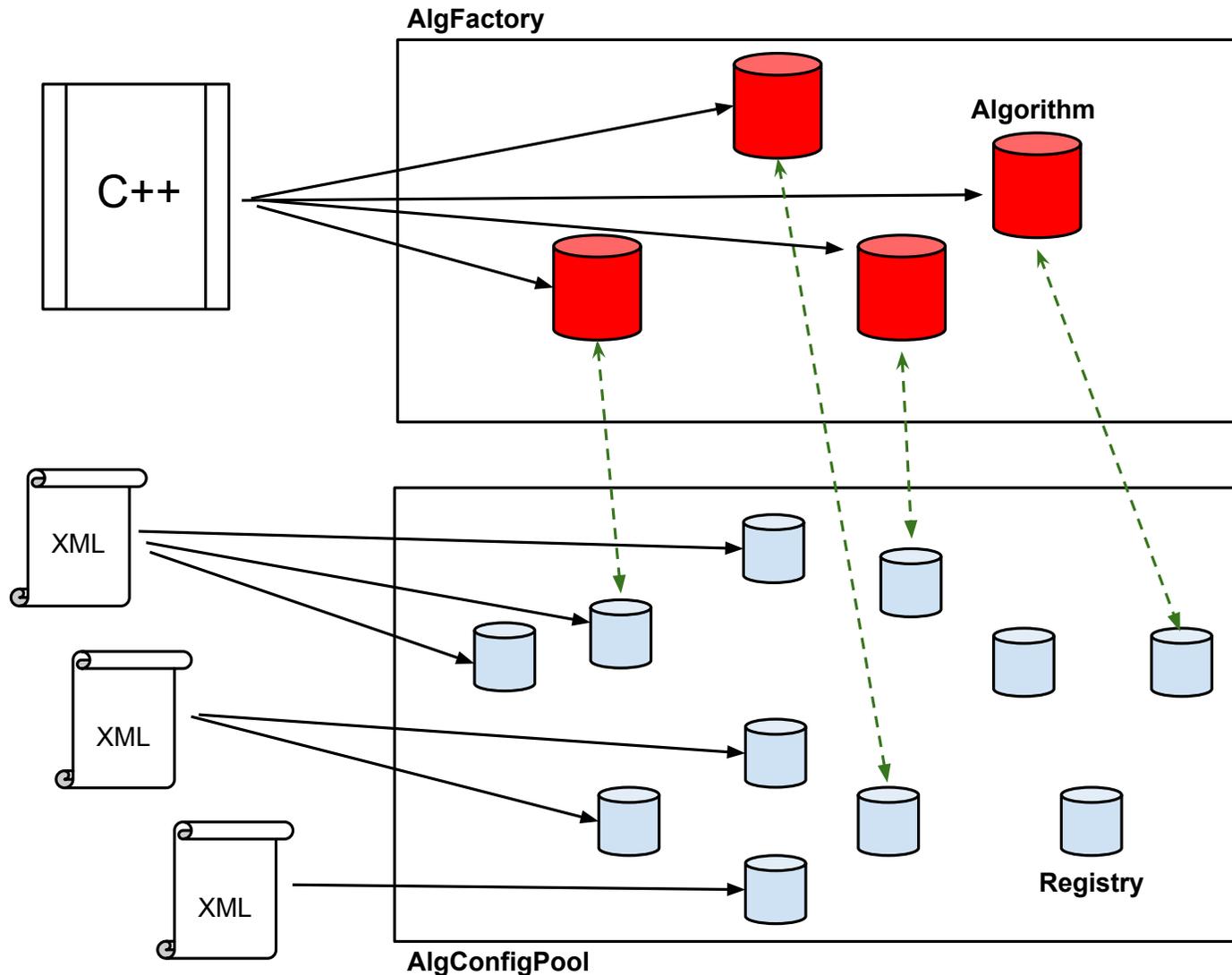
# Algorithm Factory and Configuration Pool



- One does not instantiate algorithms directly, but is using an Algorithm Factory instead (Factory design pattern).
- You specify the **algorithm name** and the **configuration name** and you get a configured '**const Algorithm \***'
- An algorithm gets instantiated when, and only if, you ask for it.
- And there is only a single instance of each algorithm you ask for (now matter how many times you ask for it).
- To do something usefull, one typecasts to an object with the desired interface methods, eg `const XSecAlgorithmI *`  
`xsec_alg_ptr = dynamic_cast<const XSecAlgorithmI *> base_ptr;`

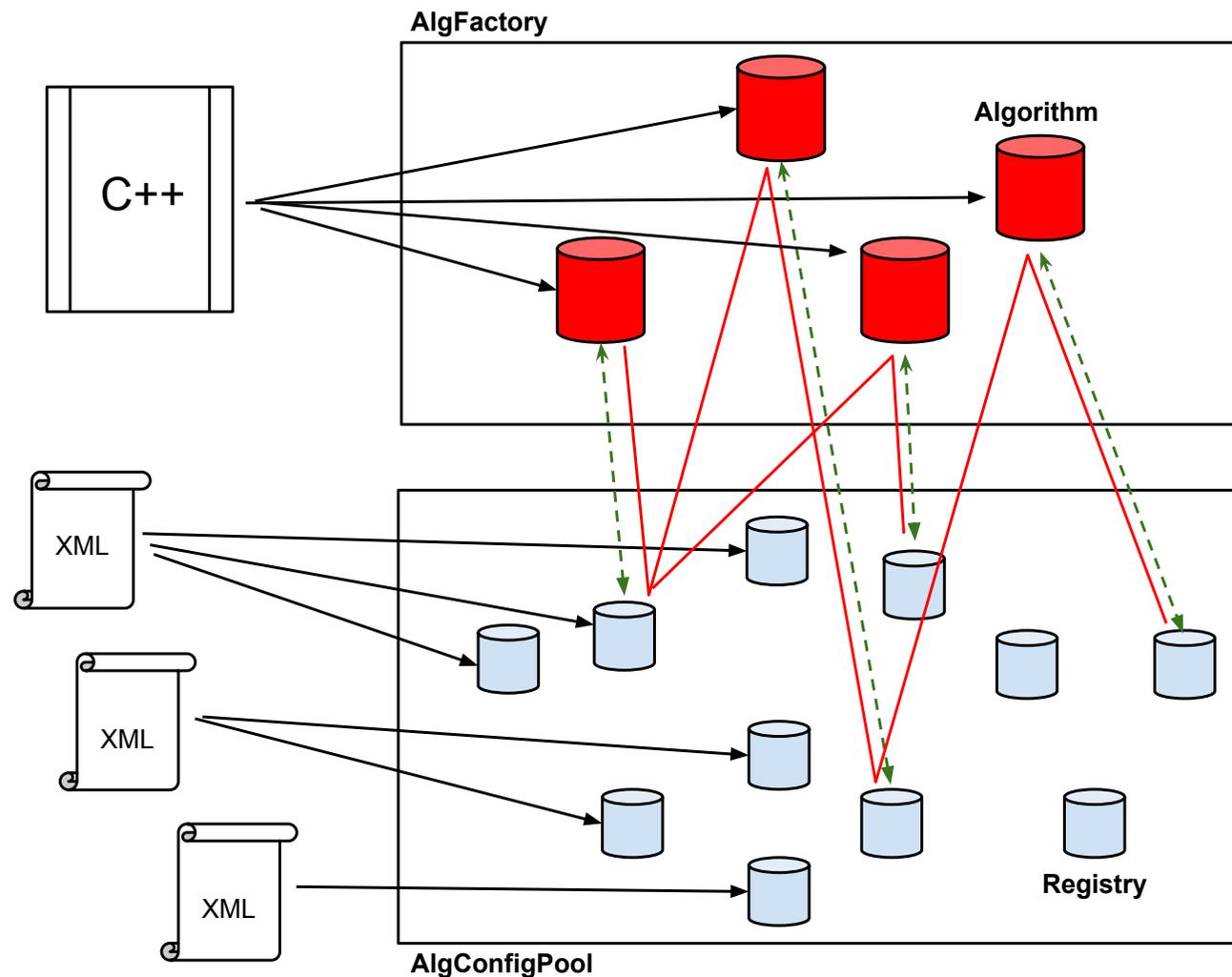
# Algorithm Factory and Configuration Pool

Instantiated algorithms and configuration registries are placed in object pools.



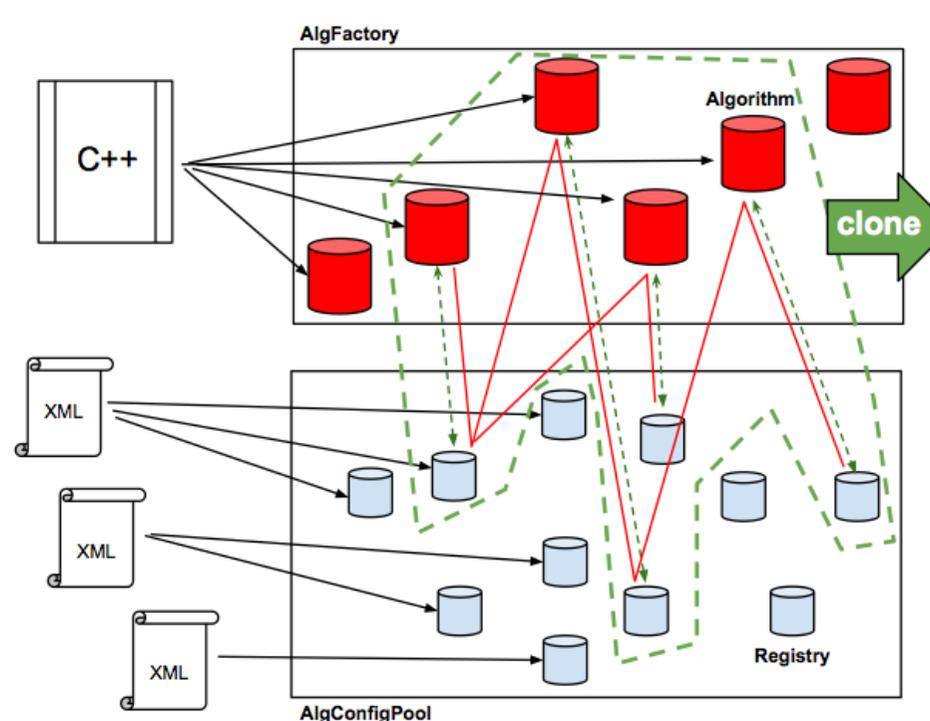
# Algorithm Factory and Configuration Pool

Running a particular configured algorithm usually involves a tree of algorithms and their corresponding configurations. Any change anywhere is propagated throughout GENIE. Important to realize the above to avoid changes which could have unintended consequences.



# Getting a monolithic algorithm block

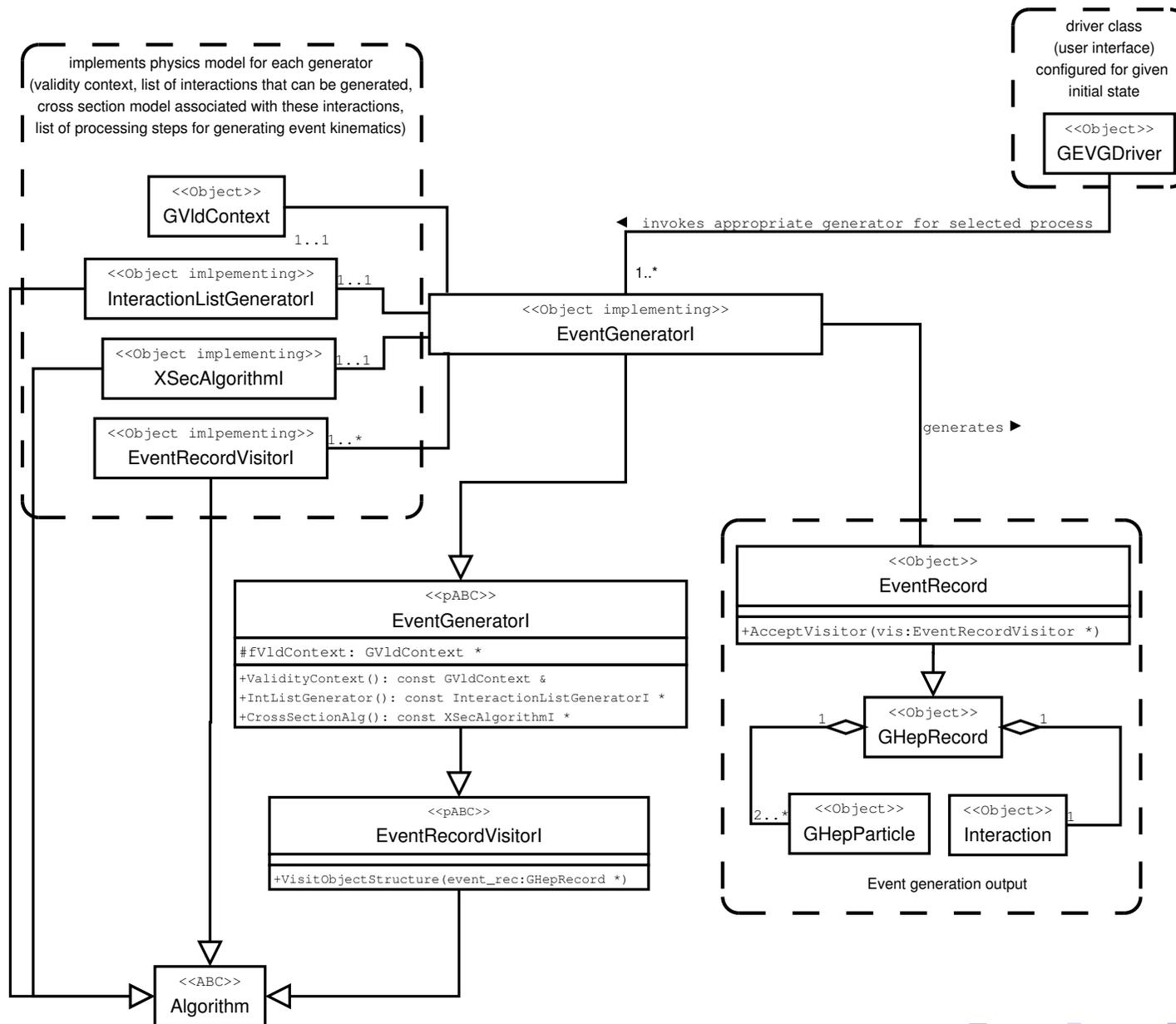
There may be use cases where it is beneficial to treat the tree of algorithms as a single algorithm with a single configuration registry (eg fitting a model to data). Such a monolithic block can be created by invoking the `AdoptSubstructure()` method at the top level algorithm. All algorithms are cloned. Instanced of sub-algorithms are stored within the top-level algorithm. The configuration registries are flattened-out and all configuration variables are placed at the registry of the top-level algorithm.



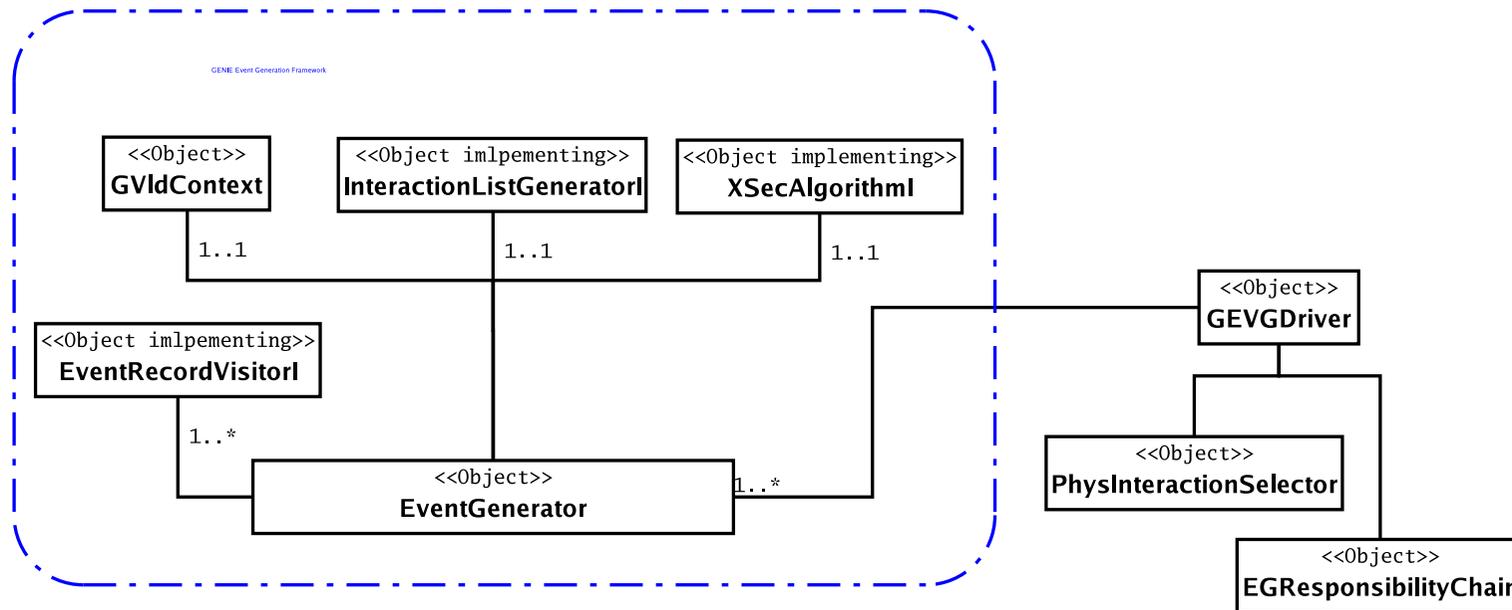
# Event Generation Framework

- Various algorithms (form factor and cross-section models, numerical algorithms, algorithms handling event record formatting, kinematics generators, particle decayers, hadronization models, intranuclear cascades) are combined together to form a generator (will see details next).
- Typically, a generator produces a given class of events (eg QE generator, resonance generator, DIS generator, etc).
- Each generator specifies
  - its validity ranges (under-used, should be usefull in future for splicing models together),
  - which processes it can generate,
  - how to compute the cross-section for these processes,
  - what are the required steps for fully simulating these processes.
- You can run a single generator on its own (but usually not for physics - explain caveats).
- For comprehensive neutrino interaction modelling, many generators need to be combined together.
- The default GENIE configuration specifies such a generator combination.
- For a given initial state, the main user interface class is called **GEVGD**river (Genie EVent Generation Driver - a provisional, temporary name that we are stuck with)

# Event Generation Framework



# Selecting an interaction and the corresponding generator



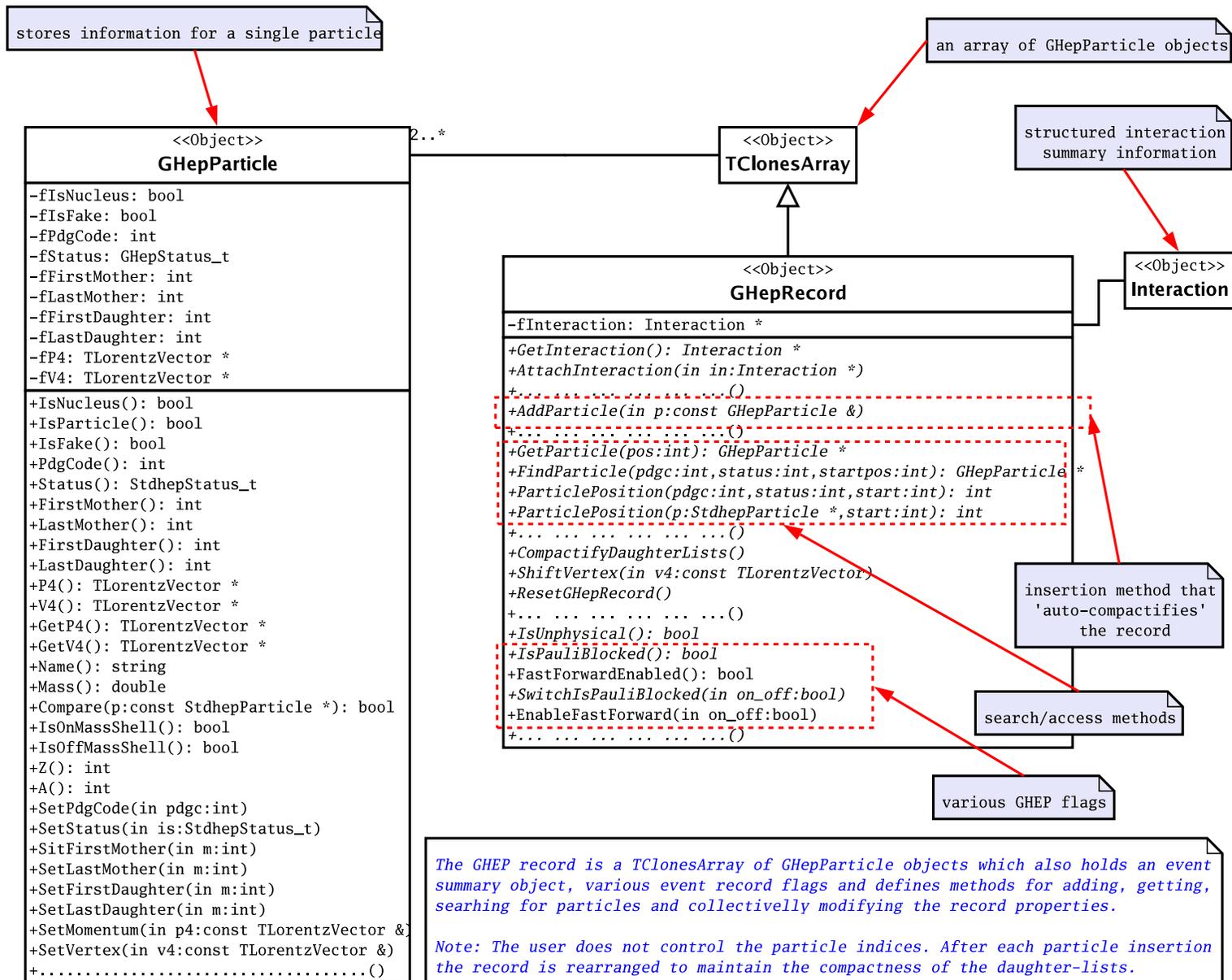
For a given initial state:

- Loop over all generators loaded in the event generation driver, query each one for their interaction list and create a map between generators and physics processes.
- Loop over processes in above map and get the corresponding generator. Through the generator get access to the cross-section model and calculate the cross-section for the given initial state.
- Once all cross-sections are computed, roll the dice and pick a process.
- Working backwards, locate the first (and only) generator which claimed it can generate kinematics for the chosen process (Chain of Responsibility pattern).
- Delegate responsibility to the chosen generator.

# Generating event kinematics

- The generator includes a list of "event record visitors" (algorithms). These are objects which sequentially "visit" and modify the event record (Visitor pattern).
- They do not communicate directly. At each stage, the event is the only record of what has been done so far.
- Each event record visitor looks at the event and modifies it (eg, Hadron transport model will look for hadrons within the nucleus and propagate them out).
- **Understanding the event structure is critical**
- Currently, the event structure is relatively consistent across all generators.
- However, we do not have a clear and well documented standard and some status codes tend to have a vague meaning and/or are relics that we still haven't shed. This has to be addressed in the near future.

# The GHEP event record



# Understanding GENIE events - An example

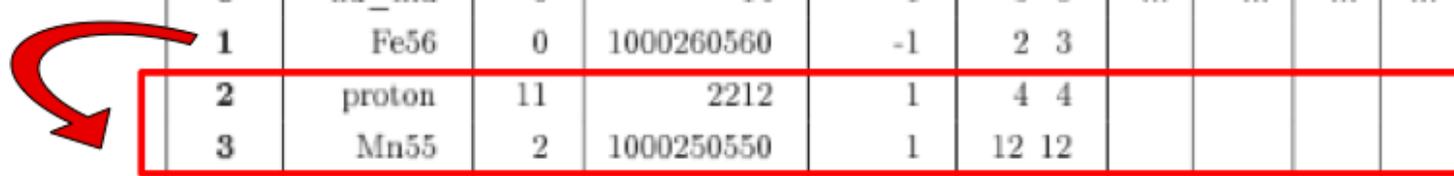
## A nu\_mu + Fe56 resonance event

initial state

Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				

# Understanding GENIE events - An example

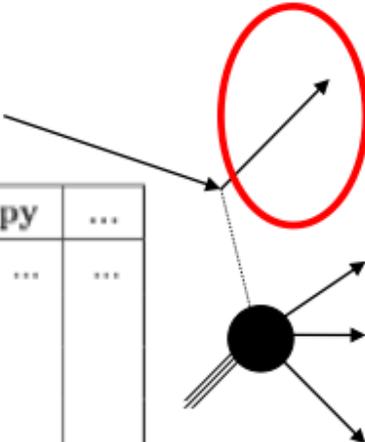
Fe56 = { hit nucleon } + { 'remnant' nucleus } = p + Mn55



Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				

# Understanding GENIE events - An example

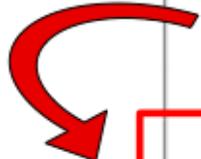
Incoming neutrino → final state primary lepton (eg. numu CC → mu-)



Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				

# Understanding GENIE events - An example

Hit proton excited to Delta++

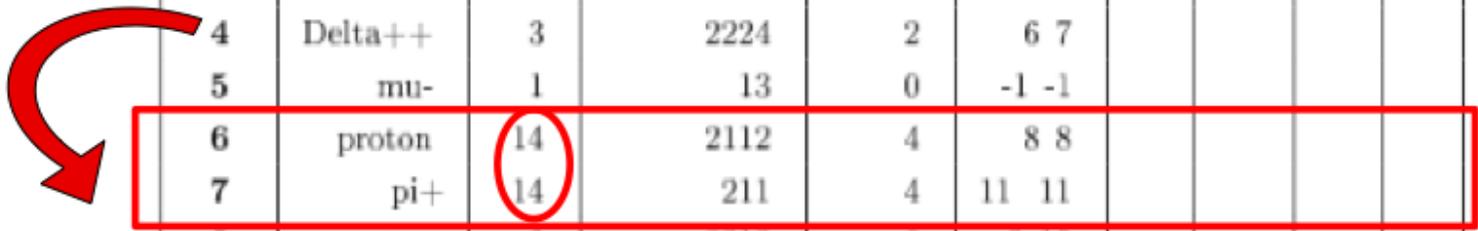


Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				

# Understanding GENIE events - An example

Delta++ decays (selected decay channel: proton pi+)

Decay happened in nuclear environment → Decay products marked as 'hadrons in the nucleus (14)'

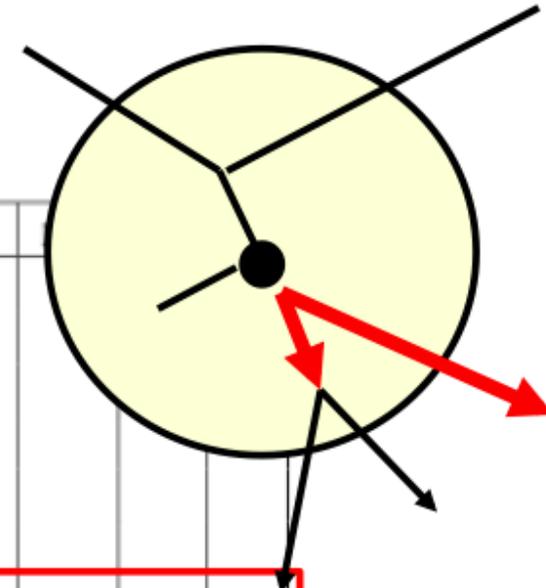


Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				

# Understanding GENIE events - An example

GENIE sees particles marked `hadrons in the nucleus (14)`  
Begin intra-nuclear hadron transport

Idx	Name	ISt	PDG	Mom	Kids	E
0	nu_mu	0	14	-1	5 5	...
1	Fe56	0	1000260560	-1	2 3	
2	proton	11	2212	1	4 4	
3	Mn55	2	1000250550	1	12 12	
4	Delta++	3	2224	2	6 7	
5	mu-	1	13	0	-1 -1	
6	proton	14	2112	4	8 8	
7	pi+	14	211	4	11 11	
8	proton	3	2212	6	9 10	
9	proton	1	2212	8	-1 -1	
10	proton	1	2212	8	-1 -1	
11	pi+	1	211	7	-1 -1	
12	HadrBlob	15	2000000002	3	-1 -1	



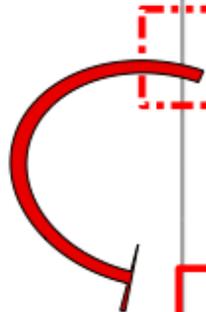
# Understanding GENIE events - An example

## Multi-nucleon knock-out

Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				

# Understanding GENIE events - An example

Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				



# Understanding GENIE events - An example

## Nuclear remnant

Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				

# Understanding GENIE events - An example

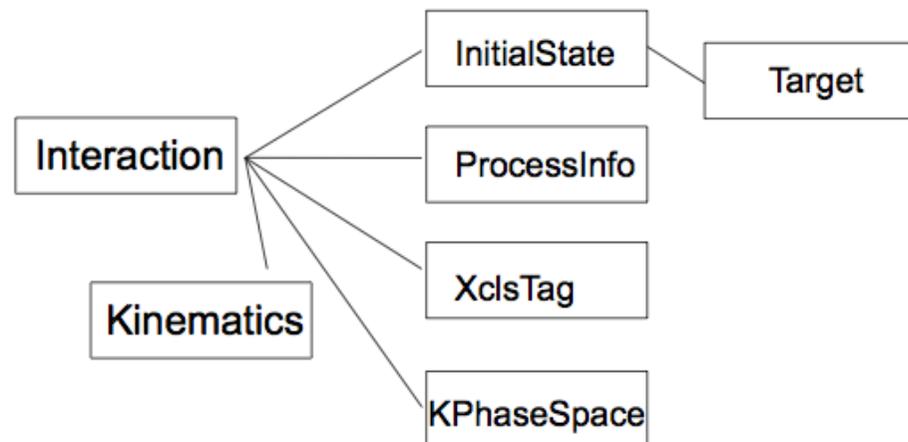
(Neutrino generator) Final state particles

To be passed-on to detector (eg Geant4-based) simulation

Idx	Name	ISt	PDG	Mom	Kids	E	px	py	...
0	nu_mu	0	14	-1	5 5	...	...	...	...
1	Fe56	0	1000260560	-1	2 3				
2	proton	11	2212	1	4 4				
3	Mn55	2	1000250550	1	12 12				
4	Delta++	3	2224	2	6 7				
5	mu-	1	13	0	-1 -1				
6	proton	14	2112	4	8 8				
7	pi+	14	211	4	11 11				
8	proton	3	2212	6	9 10				
9	proton	1	2212	8	-1 -1				
10	proton	1	2212	8	-1 -1				
11	pi+	1	211	7	-1 -1				
12	HadrBlob	15	2000000002	3	-1 -1				

# Interaction summary

- The event record also includes a summary (Interaction). A very useful object that includes key information about the event.



- Would have been tedious to have to analyze the event record every time you want to get  $Q^2$  or the neutrino energy at the hit nucleon rest frame.
- Many algorithms (eg cross-section) use an Interaction as input, not a GHEP record. One can easily invoke these algorithms for use-cases that do not involve event generation.
- Despite being a complex collection of objects can be easily instantiated using the named-constructor C++ idiom, eg  
`Interaction * ccqe = Interaction::QELCC(1000060120,2112,14); ( $\nu_\mu + (n)C^{12}$ )`
- It also plays the role of the "reaction code" used in fortran generators.
- Note main caveat: Some information about the event is available both by analyzing the GHEP record and stored in an Interaction object. It has to be the same. It will not be the same unless you explicitly take care of it.

# MC Dead-Ends

- The event record visitors apply themselves sequentially to the event record.
- Within each such module one tries things, evaluates their probability, rolls the dice and decides whether to accept them or not.
- Occasionally, based on the route already chosen by previous modules, a module may not be able to save the day and you have to **abort the event**.
- Do not exit()! This is not an error.
  - In fact, even if you encounter an error (unless it is a major one) do not exit()!
  - Better to have a 0.1% of mis-simulated events than having 0.1% of events causing 100% failure rate.
- **Throw an exception!**
- And use the **message service** properly to provide users and other developers with **valuable diagnostic information**.

# Throwing an Event Generation Exception

Assume you need to abort an event because of X reason. Do:

```
genie::exceptions::EVGThreadException e;  
e.SetReason(" All hell break loose" );  
e.SwitchOnFastForward(); or e.SwitchOnStepBack(); or e.SetReturnStep(N);  
throw e;
```

The generator will catch the exception and do as it is told:

- Could skip all further processing (which would also fail) and return an incomplete event. You may opt to reject such events or accept them and write them out in the event tree.
- Alternatively, you might go back to any step of the processing chain.
- We can store snapshots of the event record after each processing step. So an "undo" is possible.
- In principle we can return to an arbitrary point of the processing chain. In practice, we usually go back at the beginning or directly at the end.
- If you want to accept a particular type of incomplete event, make sure you flag it appropriately. GENIE includes a 16-bit error code field and you can switch on particular error codes, eg: `event->EventFlags()->SetBitNumber(kPauliBlock, true);`
- When you run GENIE, you can specify another 16-bit field mask and we use a bit-wise AND to determine which errors to ignore.

Powerfull stuff, but do not abuse if not necessary - Keep it simple!

# The Message Service

- No cout or cerr.
- Use the GENIE message service wisely to provide usefull diagnostic information.
- GENIE can produce too much print-out and usefull information might be lost if all information is sent to a single message stream.
- In production mode, all message thresholds are set to warning and important information might be lost if it is not flagged-up correctly.
- Just type:  
`LOG("stream_name", priority) << "some message";`  
where priority is any of:  
pFATAL, pALERT, pCRIT, pERROR, pWARN, pNOTICE, pInfo or pDEBUG

Do not hard-code, do not duplicate definitions (eg PDG codes), constants (eg masses), code (eg kinematic limits).

Look what is available in various packages, eg in

- BaryonResonance
- Conventions
- PDG
- Nuclear
- Utils

# Other parts of the framework

- I have described key concepts of the **core** event generation framework
- It helps GENIE in its core mission:
  - 1) *"give me an initial state (eg 1 GeV  $\nu_e$  + Ar), and I will generate events."*
- We also consider the following two as part of GENIE's mission:
  - 2) *"give me a neutrino flux and a detector geometry description, and I will generate events."*
  - 3) *"give me a simulated observable distribution and I will calculate the uncertainty on it."*
- Additional framework layers were developed for 2 and 3.
- 2 is extremely well-developed and mature, one of the main GENIE success stories. It allows experiments to generate realistic MC for complex experimental setups using off-the-shelf components.
- 3 is also well developed, functional and very useful, but with some issues to be addressed in future releases (definition of "nominal" physics should be based on meta-data stored in the generated event files, not the present default state of the generator which could be different)
- Both 2 and 3 are complex and won't discuss them right now.

Enjoy a week of GENIE coding!

Will always be available for  
questions and discussion.