EIC Project Detector-1
"We need a real name"

EPSCI
Experimental Physics Software and Computing Infrastructure

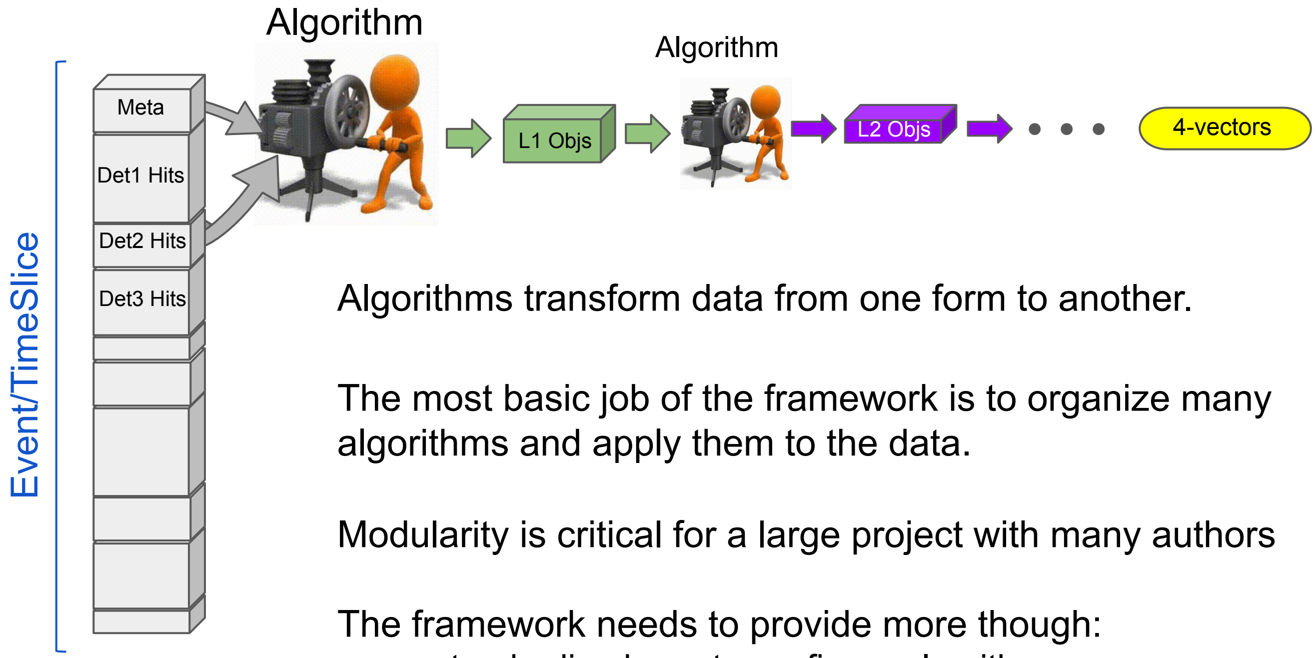
JANA2

June 29, 2022

David Lawrence JLab

EICUG/CompSW WG Joint meeting

Purpose of the “framework”



Algorithms transform data from one form to another.

The most basic job of the framework is to organize many algorithms and apply them to the data.

Modularity is critical for a large project with many authors

The framework needs to provide more though:

- standardized way to configure algorithms
- standardized control over local resources (CPUS, GPUS)
- Geometry, calibration, alignment, … services

EIC Project Detector-1
"We need a real name"

The JANA Framework

EPSCI
Experimental Physics Software and Computing Infrastructure

- JANA is a multithreaded framework project with nearly 2 decades of experience behind it
- JANA2 is a rewrite incorporating more modern coding and CS practices and improving on the original using lessons learned
  - Streaming DAQ and Heterogeneous hardware support strongly considered in redesign
- JLab is ready to commit ~1 FTE to feature development, support and implementation in the EIC software stack
  - Nathan Brei, David Lawrence, Dmitry Romanov, + others in EPSCI/EIC
  - Very interested in elevating this project to include community involvement

Projects using JANA

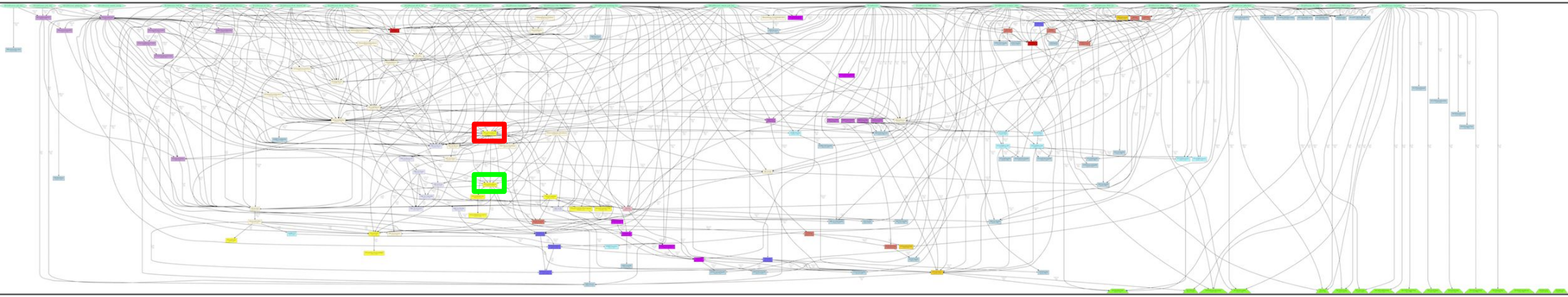
- GlueX
- INDRA-ASTRA *(near-realtime calibrations using AI/ML)*
- BDX
- TriDAS (+ERSAP) + JANA2 Streaming DAQ

Large experiments have complex call graphs

EIC Project Detector-1
"We need a real name"

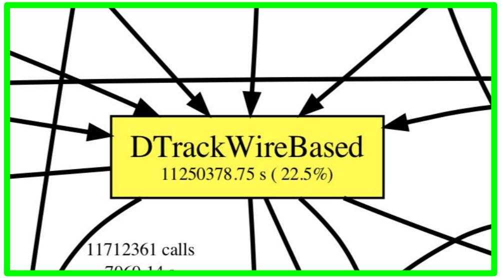
EPSCI
Experimental Physics Software and Computing Infrastructure

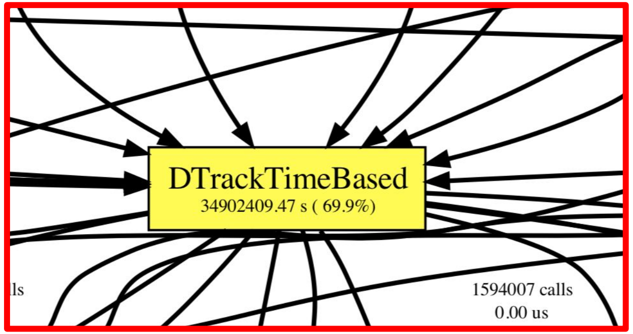
GlueX Reconstruction - automated rendering via janadot plugin



Run 42513:

Physics Production mode Trigger: FCAL_BCAL_PS_m9.conf
setup: hd_all.tsg
0/90 PERP 90
JD70-100 58um
TPOL Be 75um
beam looks stable



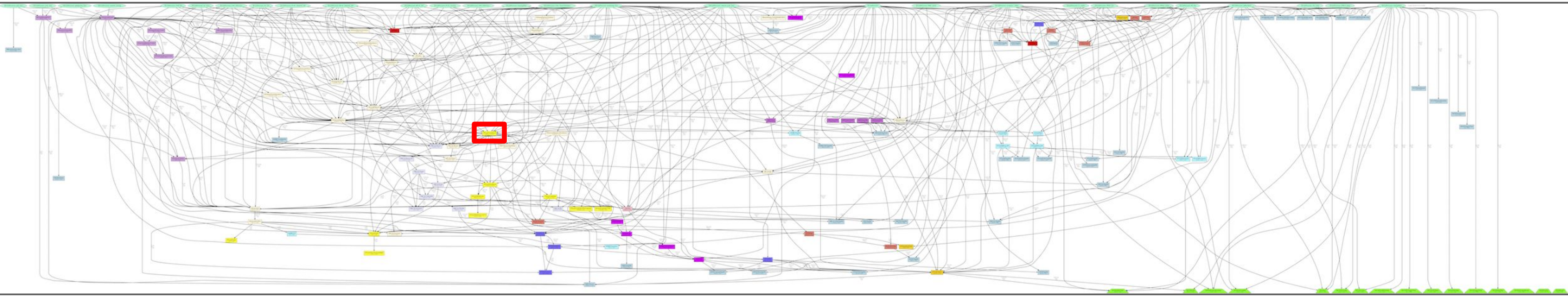


EIC Project Detector-1
"We need a real name"

Large experiments have complex call graphs

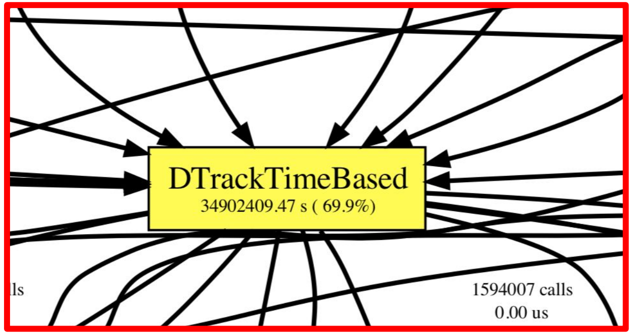
EPSCI
Experimental Physics Software and Computing Infrastructure

GlueX Reconstruction - automated rendering via janadot plugin



Modular design:

- Factories (algorithms) need to know what they depend on
- Factories do ***not*** need to know what depends on them
- Dependencies do ***not*** need to be specified at higher level

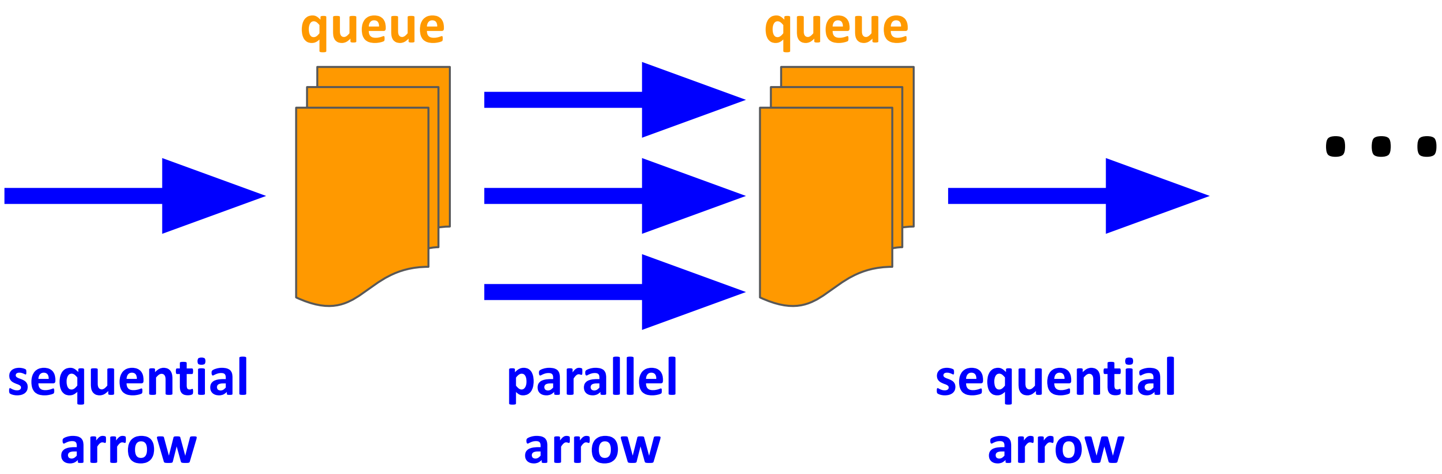


EIC Project Detector-1
"We need a real name"

JANA2 arrows separate Sequential and Parallel tasks

EPSCI
Experimental Physics Software and Computing Infrastructure

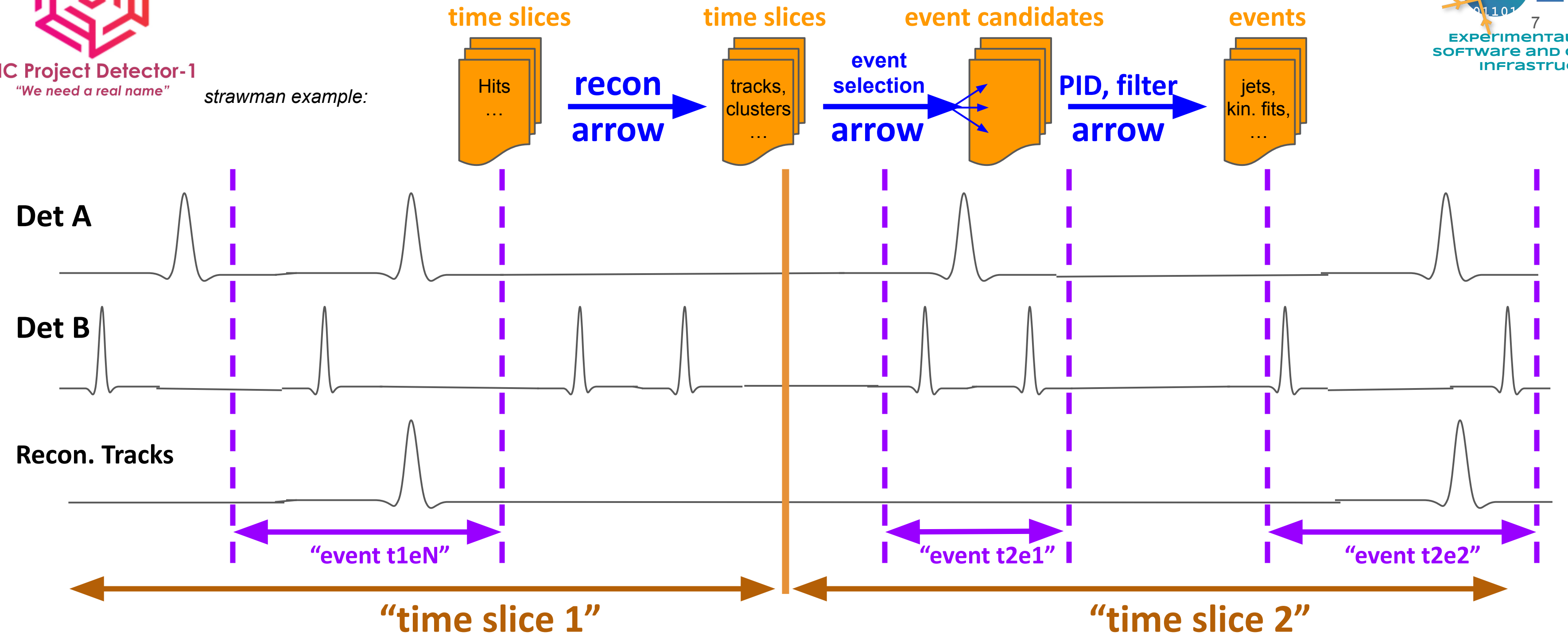
- CPU intensive event reconstruction will be done as a parallel arrow
- Other tasks (e.g. I/O) can be done as a sequential arrow
- Fewer locks in user code allows framework to better optimize workflow



Streaming Data

EIC Project Detector-1
"We need a real name"

strawman example:



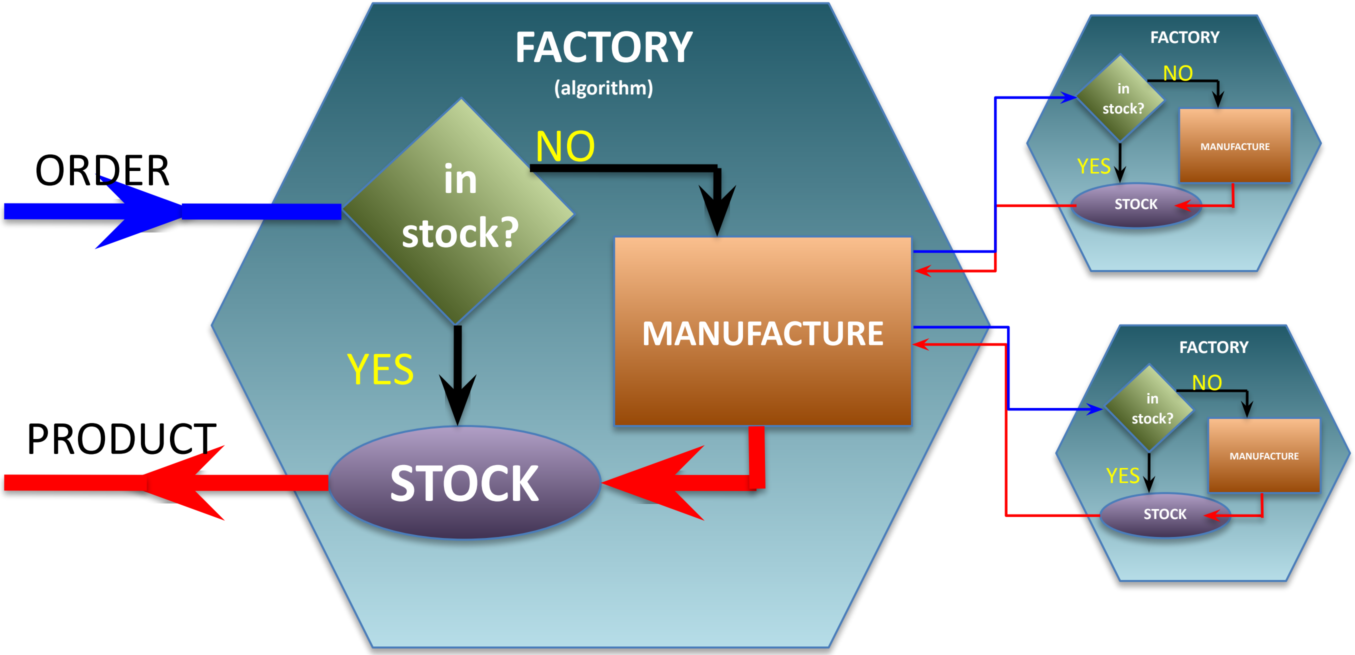
EPSCI
Experimental Physics Software and Computing Infrastructure

- Stream comes in the form of large time slices which may contain many events
- Arrow/queue system naturally supports one-to-many transformations
- Used in (ERSAP+) TriDAS + JANA2 system in Hall-B/Hall-D
- Used in INDRA-ASTRA AI/ML near-realtime calibration project

EIC Project Detector-1
"We need a real name"

EPSCI
Experimental Physics Software and Computing Infrastructure

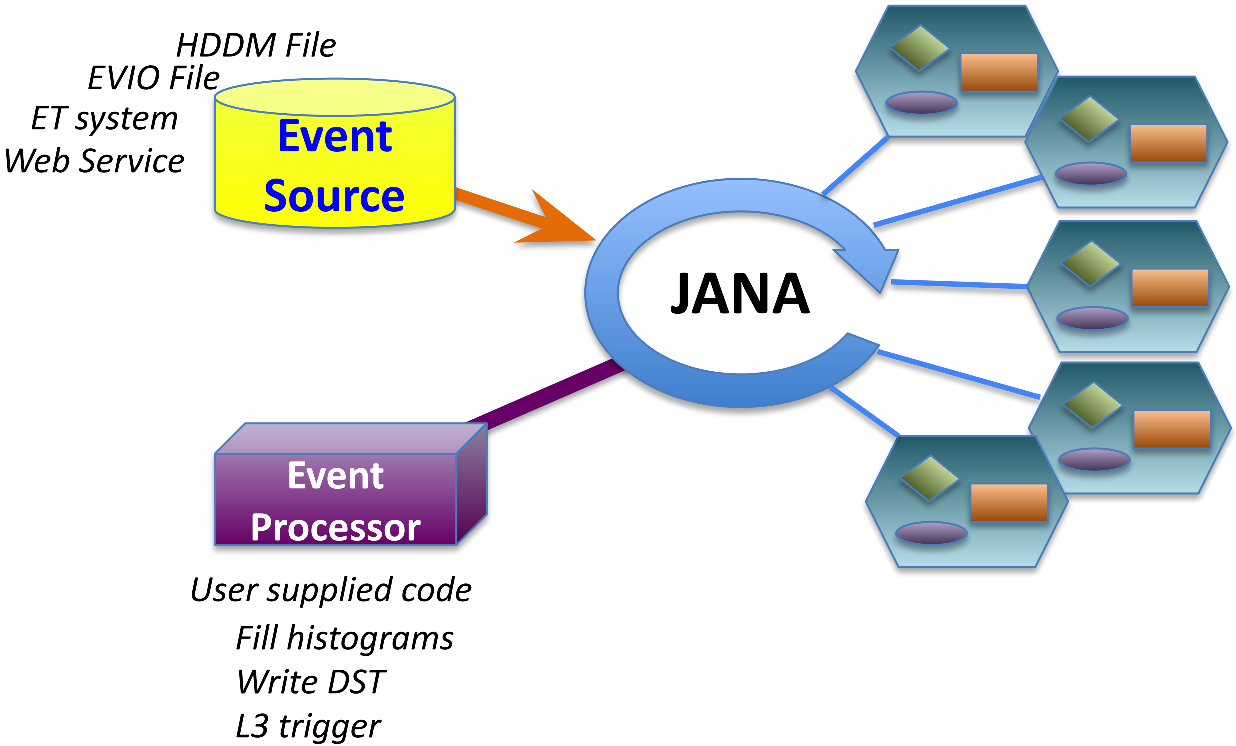
Factory Model



Data on demand = Don't do it unless you need it
Stock = Don't do it twice

Conservation of CPU cycles!

Complete Event Reconstruction in JANA



Framework has a layer that directs object requests to the factory that completes it

Multiple algorithms (factories) may exist in the same program that produce the same type of data objects

This allows the framework to easily redirect requests to alternate algorithms specified by the user at run time

Multi-threading

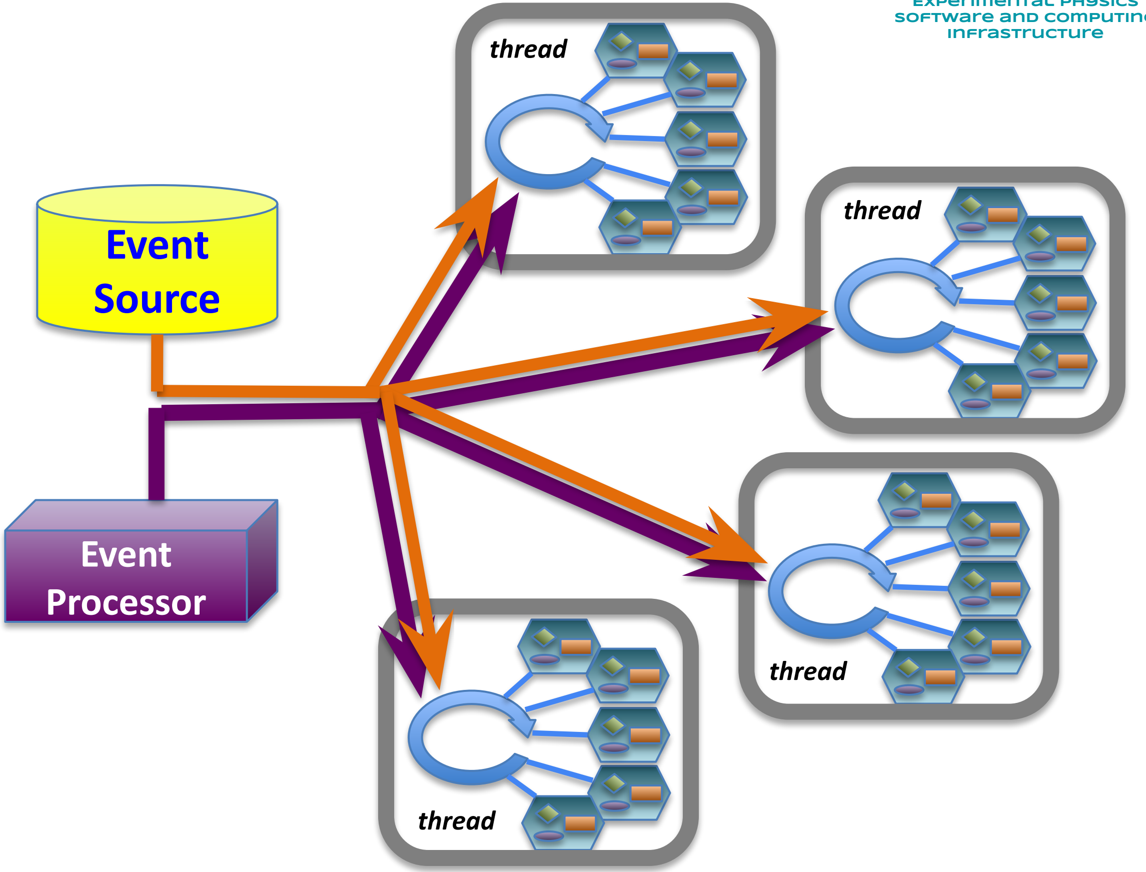
EIC Project Detector-1
"We need a real name"

EPSCI
Experimental Physics Software and Computing Infrastructure

- *Each thread has a complete set of factories making it capable of completely reconstructing a single event/slice*

- *Factories only work with other factories in the same thread **eliminating the need for expensive mutex locking** within the factories*

- *All events are seen by all Event Processors (multiple processors can exist in a program)*



Basic data access

```
auto tracks = jevent->Get<DTrack>();

for(auto t : tracks){

  // ... do something with const DTrack* t

}
```

n.b. `std::vector<const DTrack*> tracks;`

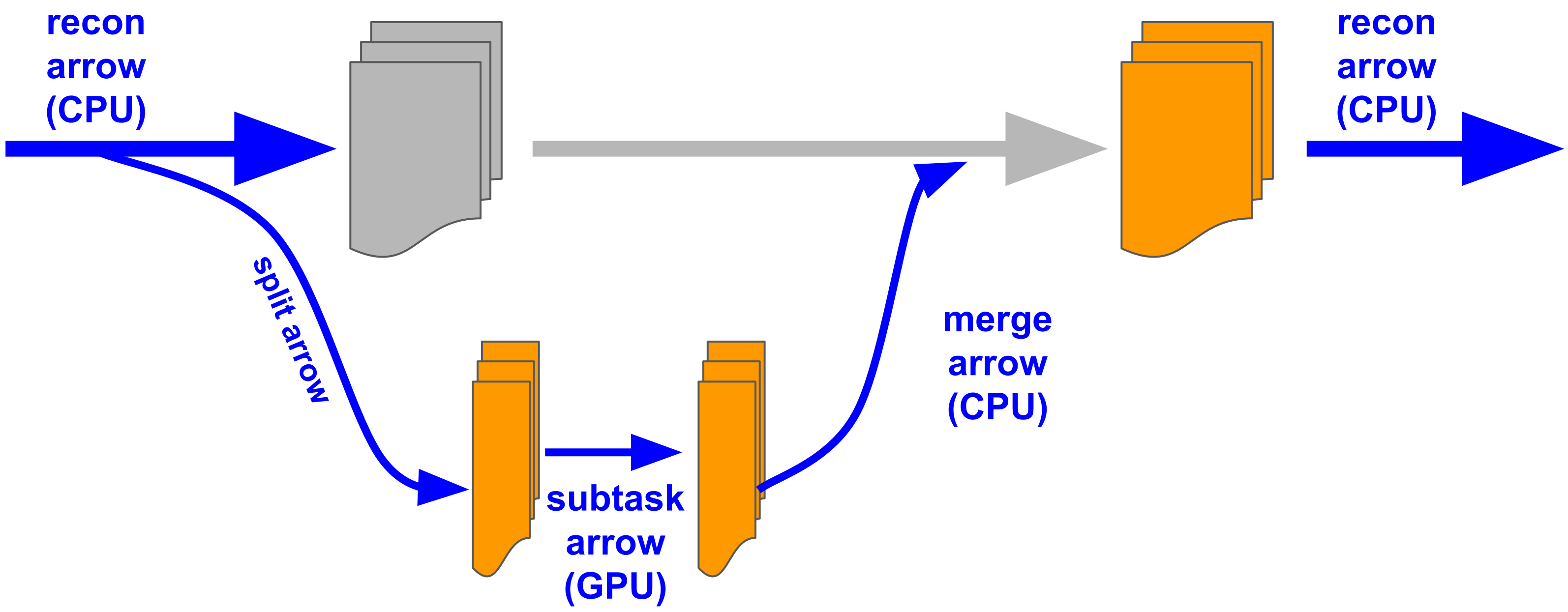
Boilerplate code generation

```
> jana-generate.py
Usage: jana-generate.py [-h|--help] [type] [args...]
  type: JObject JEventSource JEventProcessor RootEventProcessor JEventProcessorTest JFactory Plugin Project
```

```
> jana-generate.py --help
…
Plugin
Create a code skeleton for a plugin in its own directory. Takes the following positional arguments:
          name            The name of the plugin, e.g. "trk_eff" or "TrackingEfficiency"
         [is_standalone]  Is this a new project, or are we inside the source tree of an existing CMake project? (default=True)
         [is_mini]        Reduce boilerplate and put everything in a single file? (default=True)
         [include_root]   Include a ROOT dependency and stubs for filling a ROOT histogram? (default=True)

       Example: `jana_generate.py Plugin TrackingEfficiency 1 0 0`
…
```

```
> jana-generate.py Plugin DaveTest
> ls DaveTest/
CMakeLists.txt  DaveTest.cc
> mkdir DaveTest/build
> cd DaveTest/build/
> cmake ..
…
> make install
[ 50%] Building CXX object CMakeFiles/DaveTest_plugin.dir/DaveTest.cc.o
[100%] Linking CXX shared library DaveTest.so
[100%] Built target DaveTest_plugin
Install the project...
-- Install configuration: ""
-- Installing: /Users/davidl/builds/JANA2/JANA2/plugins/DaveTest.so
```

Heterogeneous Hardware Support

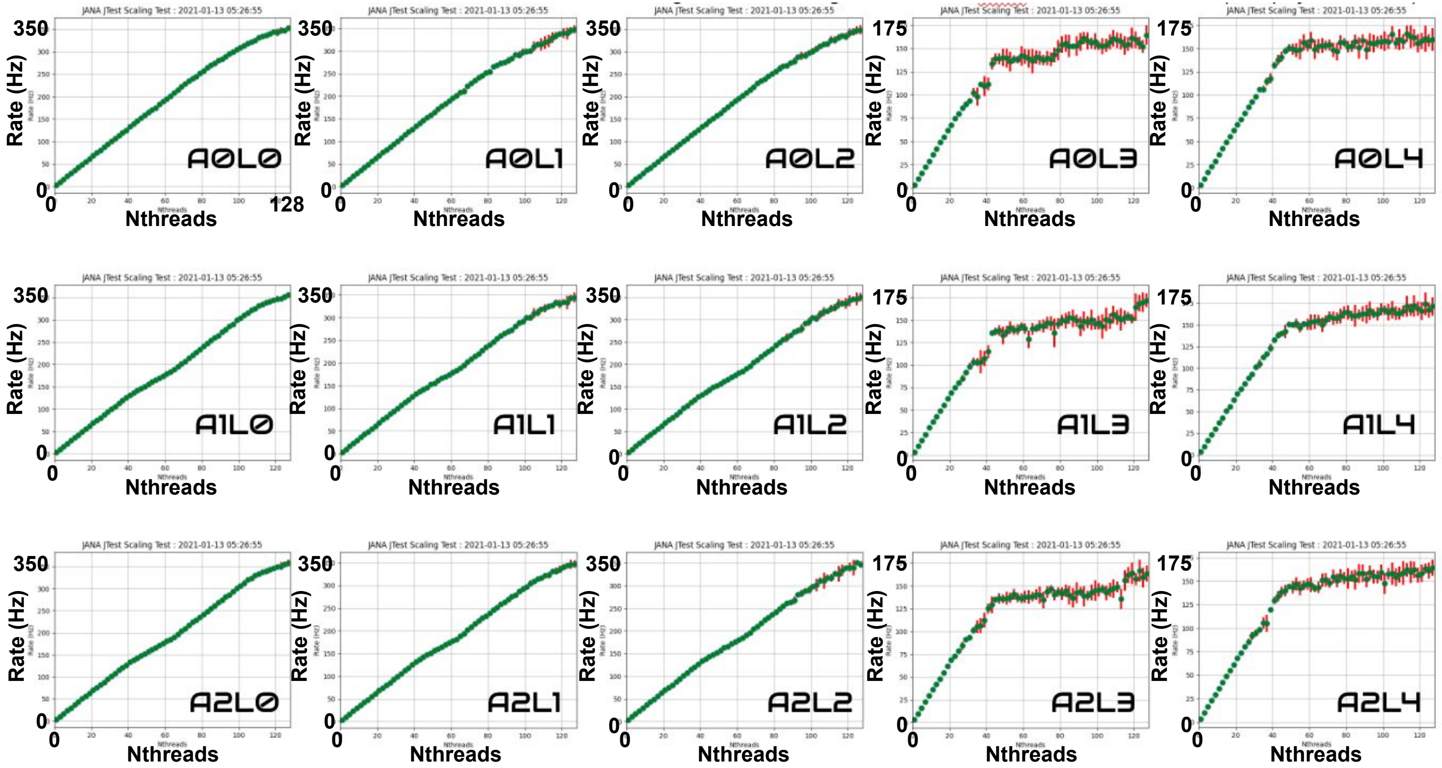


EIC Project Detector-1
"We need a real name"

EPSCI
Experimental Physics Software and Computing Infrastructure

Multiple Affinity and Locality strategies

OS, chip type, memory architecture, and nature of job all can affect which model yields optimal performance



JANA2 Scaling test: PSC Bridges-2 RM Two AMD EPYC 7742 CPUS (128 physical cores)

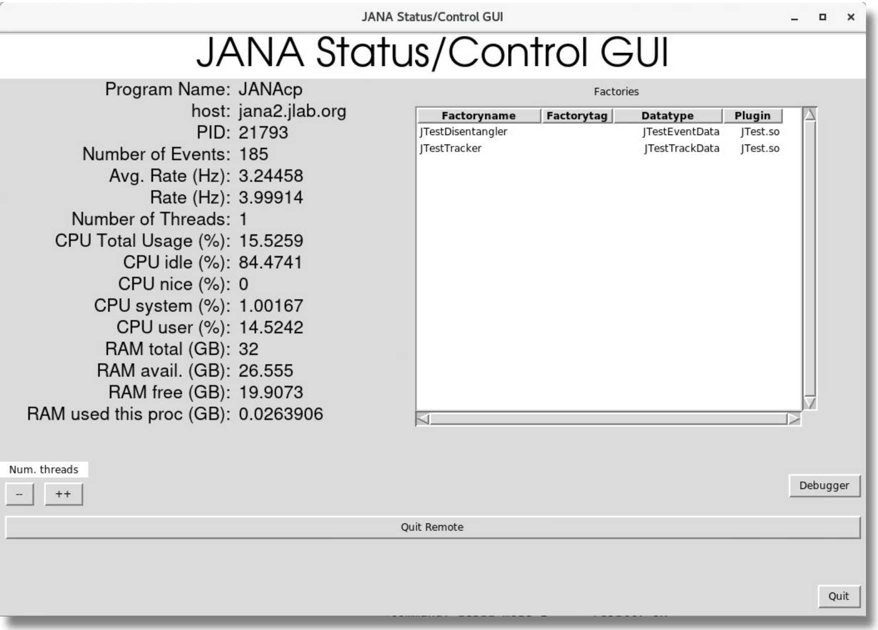
```
enum class
AffinityStrategy {
      None,
      MemoryBound,
      ComputeBound };

enum class
LocalityStrategy {
      Global,
      SocketLocal,
      NumaDomainLocal,
      CoreLocal,
      CpuLocal };
```

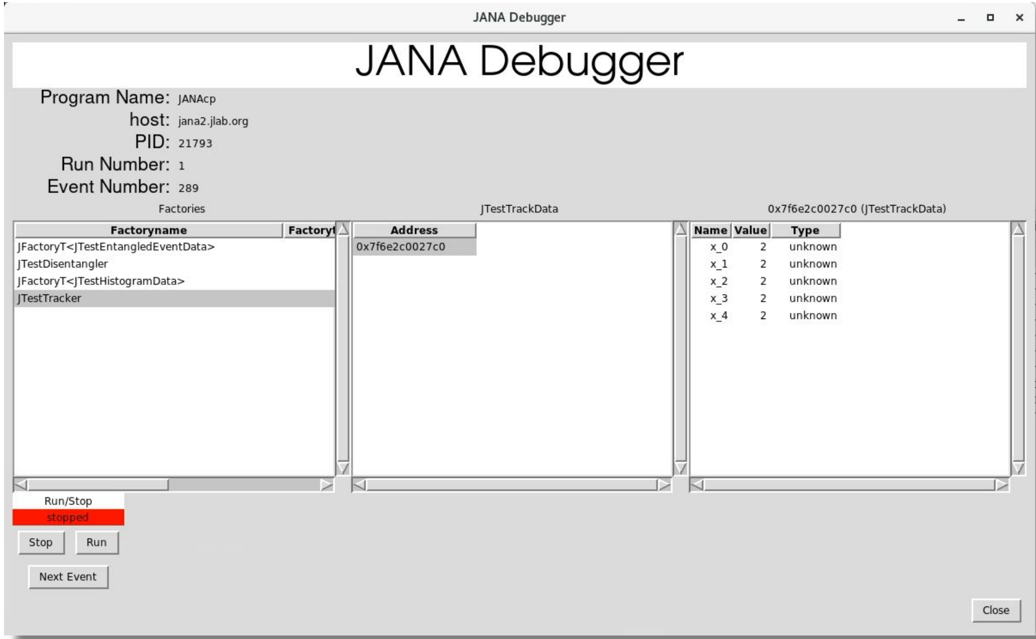
Configurable at run time via Config. Parameters

EIC Project Detector-1
"We need a real name"

Inspection Tools

EPSCI
Experimental Physics Software and Computing Infrastructure






```
> jana -Pplugins=JTest,janacontrol
```

Add *janacontrol* plugin to any process

```
> jana-control.py [--host host] [--port port]
```

Run GUI from remote (or same) node

EIC Project Detector-1
"We need a real name"

JANA Command Line Debugging w/ gdb

EPSCI
Experimental Physics Software and Computing Infrastructure

davidl@jana2:JANA

File Edit View Search Terminal Help

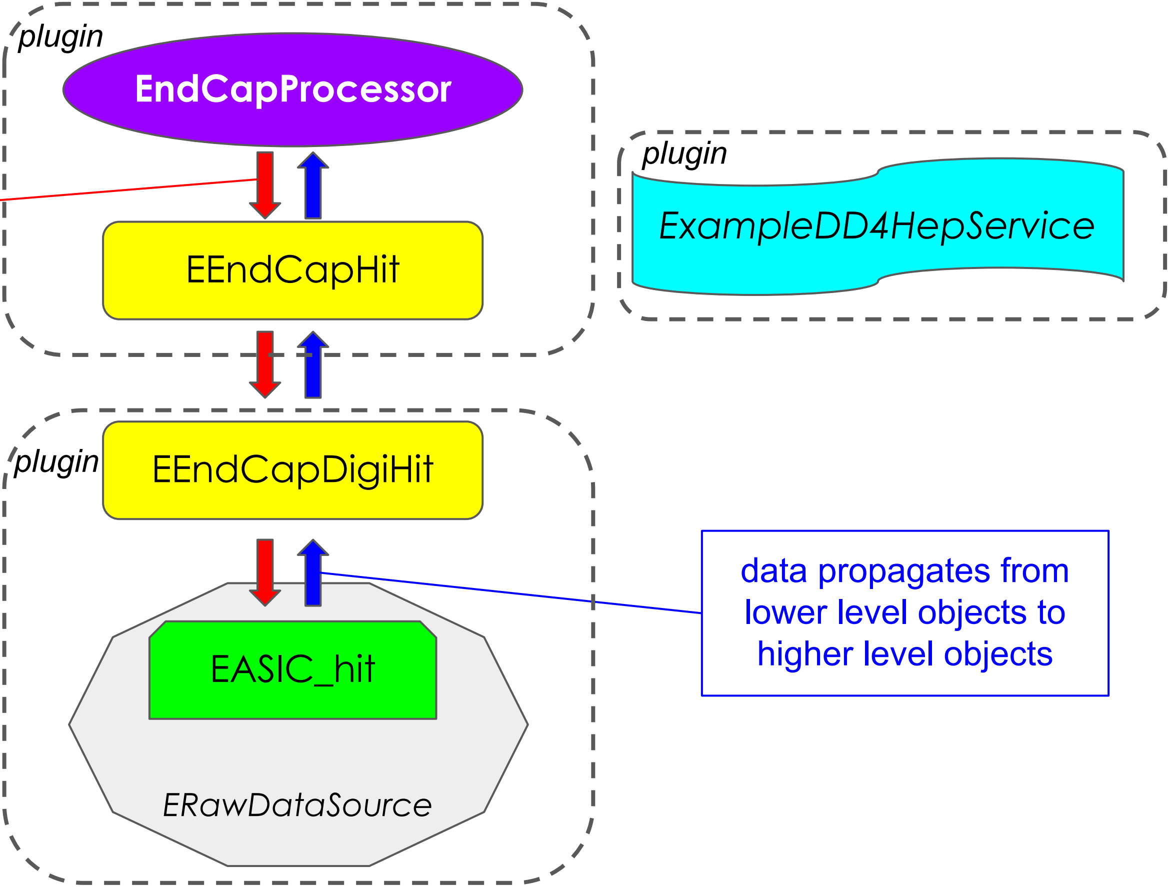
```
Class name:    JTestParser
Sequential:    0

JANA: [INFO] Status: 0 events processed  0.0 Hz (0.0 Hz avg)


JANA: h
  ----------------------------------------
  Available commands
  ----------------------------------------
  pe    PrintEvent
  pf    PrintFactories [filter_level <- {0,1,2,3}]
  pfd   PrintFactoryDetails fac_idx
  po    PrintObjects fac_idx
  po    PrintObject fac_idx obj_idx
  pfp   PrintFactoryParents fac_idx
  pop   PrintObjectParents fac_idx obj_idx
  poa   PrintObjectAncestors fac_idx obj_idx
  vt    ViewAsTable
  vj    ViewAsJson
  x     Exit
  h     Help
  ----------------------------------------

JANA: p
```

Certain JANA methods are written with the intention of being called from debugger.

This allows easier browsing from the framework point of of view.

Example with Geometry Service

https://github.com/faustus123/EIC_JANA_Example



requests start with higher level objects and propagate to lower level objects

data propagates from lower level objects to higher level objects

Wouter suggested example:

"…[take] a collection of hits and selecting those hits that are on a particular endcap tracking detector and have a position outside a minimum radial range."

JFactory_EEndCapHit


```cpp
#ifndef _JFactory_EEndCapHit_h_
#define _JFactory_EEndCapHit_h_

#include <JANA/JFactoryT.h>
#include <ExampleDD4HepService/ExampleDD4HepService.h>
#include "EEndCapHit.h"

class JFactory_EEndCapHit : public JFactoryT<EEndCapHit> {

    // Insert any member variables here

public:
    JFactory_EEndCapHit();
    void Init() override;
    void ChangeRun(const std::shared_ptr<const JEvent> &event) override;
    void Process(const std::shared_ptr<const JEvent> &event) override;

protected:
    double min_radius;

    const ExampleDD4HepService *geomservice=nullptr;

};

#endif // _JFactory_EEndCapHit_h_
```

```cpp
void JFactory_EEndCapHit::Init() {
    auto app = GetApplication();

    // Just for fun, create a configuration parameter named
    // EndCap:min_radius so we can set the threshold at run time.
    min_radius = 15.0;
    app->SetDefaultParameter("EndCap:min_radius", min_radius, "The mini

    /// Acquire geometry service pointer (see ExampleDD4HepService plugin)
    geomservice = app->GetService<ExampleDD4HepService>().get();
}
```

boilerplate

added for this example

EIC Project Detector-1

"We need a real name"

JFactory_EEndCapHit::Process

EXPERIMENTAL PHYSICS SOFTWARE AND COMPUTING INFRASTRUCTURE

```
//------------------------
// Process
//------------------------
void JFactory_EEndCapHit::Process(const std::shared_ptr<const JEvent> &event) {

    /// JFactories are local to a thread, so we are free to access and modify
    /// member variables here. However, be aware that events are _scattered_ to
    /// different JFactory instances, not _broadcast_: this means that JFactory
    /// instances only see _some_ of the events.

    // The EEndCapDigiHit objects are made by a factory in the EICRawData plugin.
    // That factory uses the low-level EASIC_hit objects coming from the event source
    auto endcapdigihits = event->Get<EEndCapDigiHit>();

    // Loop over the EEndCapDigiHit objects and create calibrated hits
    // objects with geometry info.
    std::vector<EEndCapHit *> hits;
    for( auto digihit : endcapdigihits ){

        auto pos = geomservice->GetVTXPixelLocation( digihit->layer, digihit->chip, digihit->pixel );
        auto r = pos.Perp();
        if( r > min_radius ){

            auto hit = new EEndCapHit();
            hit->x = pos.X();
            hit->y = pos.Y();
            hit->z = pos.Z();
            hit->t = ((double)digihit->t - 125.0)*2.50E-1;  // Here we would apply calibrations read from DB
            hits.push_back(hit);
        }
    }

    /// Publish outputs
    Set(hits);

    // n.b. if we created additional types of objects we could also add them to the event using event->Insert() )
}
```

ExampleDD4HepService

EIC Project Detector-1
"We need a real name"

EPSCI
Experimental Physics Software and Computing Infrastructure

```
class ExampleDD4HepService: public JService {

public:
    // Constructor
    ExampleDD4HepService()=default;

    // The geometry service needs to be sensitive to the exact data being processed since subtle
    // alignment changes or even significant changes to the detector could appear between one
    // data set and the next. The most versatile system would allow data from multiple different
    // geometry definitions to exist at the same time.
    //
    // For this to return the correct geometry, it needs information from the data stream itself
    // on when it was acquired so it can access the correct DB. I do not try and add that
    // complication here right now. I do demonstrate though that the JEvent reference would be
    // passed in so that the needed info can be extracted. Note that this should not be called for
    // every event, but rather from the ChangeRun method of a factory or processor indicating a
    // new calibration region of the stream has been reached.
    const dd4hep::Assembly* GetDD4hepAssembly(const std::shared_ptr<const JEvent> &event) const {

        // Retrieve the correct Assembly based on when the given
        // JEvent was acquired.

        return _assembly;
    }

    // There is a lot of freedom in how this class could be organized. One is to simply provide a
    // reference to the DD4hep Assembly object as above and let all of the algorithms speak "DD4hep".
    // A more practical approach would be to augment that with some dedicated methods that answer
    // common questions about the geometry for specific detectors. Here is an example of this:
    TVector3 GetVTXPixelLocation( int layer, int chip, int pixel ) const {

        // This is where the code to extract the location information given the layer,chip, and pixel
        // values would reside. This could either be directly from the dd4hep reference or from some
        // cached value.
        assert( layer>=1 && layer<=9 );

        double x = (double)chip*2.7;  // Totally unrealistic. Just for demo
        double y = (double)pixel*1.2; // Totally unrealistic. Just for demo
        double z = z_layer[layer-1];  // Lookup table (this should actually be close to correct!)

        return TVector3( x, y, z);
    }

private:
    dd4hep::Assembly *_assembly = nullptr;
    double z_layer[9] = {-106.0, -79.0, -52.0, -25.0, 25.0, 49.0, 73.0, 106.0, 125.0};

};
```

Service is added to application with single line:

```
extern "C" {
    void InitPlugin(JApplication *app) {
        InitJANAPlugin(app);
        app->ProvideService( std::make_shared<ExampleDD4HepService>() );
    }
}
```

EIC Project Detector-1

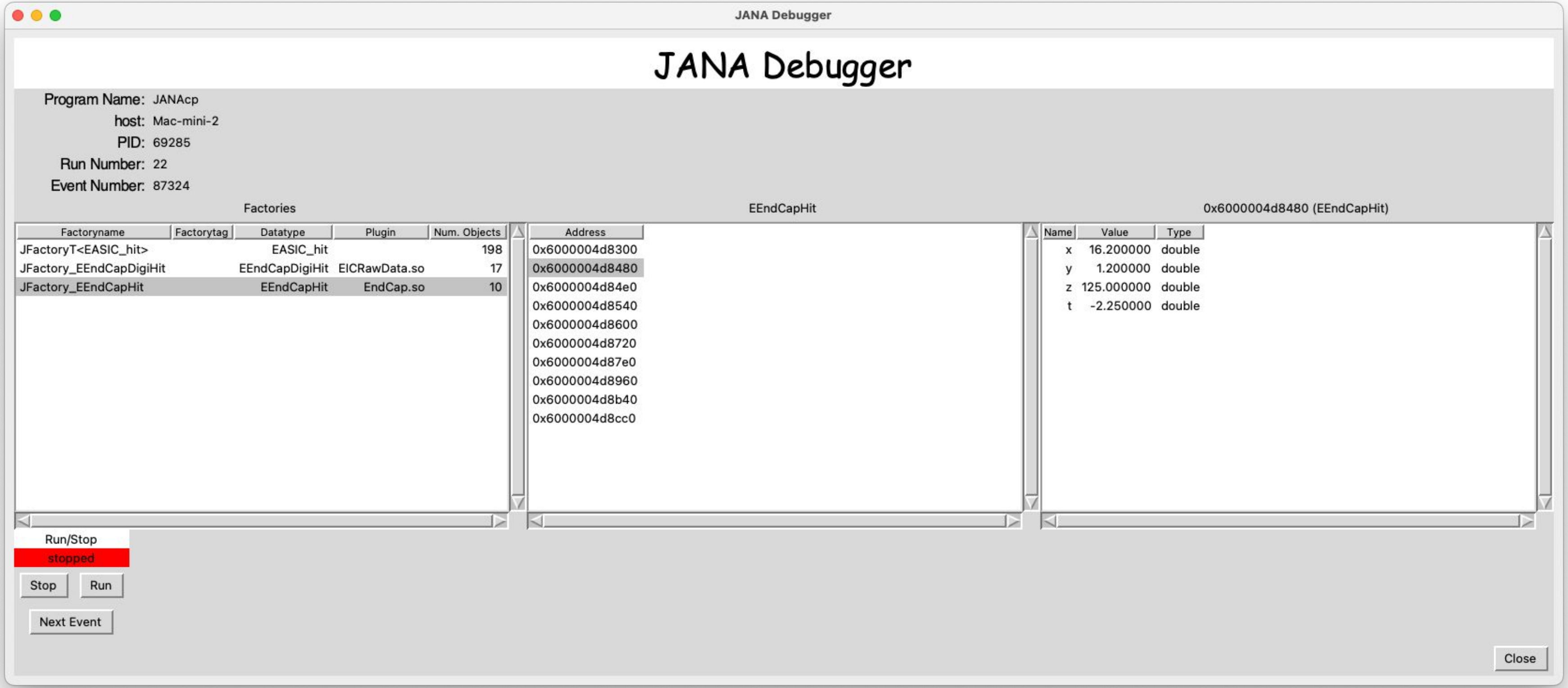
"We need a real name"

Example

https://github.com/faustus123/EIC_JANA_Example

EPSCI

Experimental Physics Software and Computing Infrastructure



EIC Project Detector-1
"We need a real name"

Summary

EXPERIMENTAL PHYSICS SOFTWARE AND COMPUTING INFRASTRUCTURE

- JANA is a multithreaded framework project with nearly 2 decades of experience behind it

- JANA2 is a rewrite incorporating more modern coding and CS practices and improving on the original using lessons learned
  - Streaming DAQ and Heterogeneous hardware support strongly considered in redesign

- JLab is a **partner lab** in the EIC project and is ready to commit **~1 FTE** to feature development, support and implementation in the EIC software stack
  - Nathan Brei, David Lawrence, Dmitry Romanov, + others in EPSCI/EIC
  - Very interested in elevating this project to include community involvement

Github: https://github.com/JeffersonLab/JANA2
Documentation: https://jeffersonlab.github.io/JANA2/
Example project: https://github.com/faustus123/EIC_JANA_Example

Publications:

https://arxiv.org/abs/2202.03085 *Streaming readout for next generation electron scattering experiments*
https://doi.org/10.1051/epjconf/202125104011 *Streaming Readout of the CLAS12 Forward Tagger Using TriDAS and JANA2*
https://doi.org/10.1051/epjconf/202024501022 *JANA2 Framework for Event Based and Triggerless Data Processing*
https://doi.org/10.1051/epjconf/202024507037 *Offsite Data Processing for the GlueX Experiment*
https://iopscience.iop.org/article/10.1088/1742-6596/119/4/042018 *Multi-threaded event reconstruction with JANA*
https://pos.sissa.it/070/062 *Multi-threaded event processing with JANA*
https://iopscience.iop.org/article/10.1088/1742-6596/219/4/042011 *The JANA calibrations and conditions database API*
https://iopscience.iop.org/article/10.1088/1742-6596/1525/1/012032 *JANA2: Multithreaded Event Reconstruction*

Requirements

- *The reconstruction framework must be able to run on both simulated events and real data. Even if there may be algorithms that use truth information (or even require truth information, initially), the reconstruction framework itself should allow for running without truth information.*
  JANA's ***factory tag*** mechanism can be used to tag "TRUTH" versions of objects. The tagged versions of objects may be requested programmatically or on a global scale at runtime via configuration parameters. Both the TRUTH tagged and the un-tagged versions of the objects may coexist.

- *The reconstruction framework must be able to take advantage of heterogeneous computing resources (multiple cores, GPUs, etc).*
  JANA's main purpose for existence was to provide multi-threaded event reconstruction and the entire design of the framework grows from that. Sub-tasks were added in JANA2 specifically to add additional heterogeneous support.

- *The reconstruction framework must encourage modular approaches to algorithm development, using defined interface layers.*
  JANA has a set of base classes that define the interface. Furthermore, the emphasis on a factory having one primary class of object as its output encourages users to implement a more modular design. e.g. Track seeds can be produced in one factory and fully fit tracks in another allowing the seed finding algorithm to be easily swapped. The framework also allows for both types of objects to be produced in a single factory, but this design encourages the code designer to break that up into smaller modules instead.

Requirements

EIC Project Detector-1
"We need a real name"

EPSCI
Experimental Physics Software and Computing Infrastructure

- *Algorithms must be implemented using the selected data model, and ensure that data (event data, geometry description, and algorithm parameters) are kept separate from the algorithm itself.*
  JANA supports this style of programming. The algorithm parameters (formally *Configuration Parameters* in JANA) can be set via config file or command line argument and a centrally available to all factories. Furthermore, the implementation allows new configuration parameters to be easily deep in a factory's user code, yet still be accessible to all JANA objects.

  The event data is managed by the framework. Geometry description is provided by a JANA Service that gives access to the underlying geometry package (e.g. DD4hep).

- *Algorithms must be implemented in the framework independently from any scheduling strategies; an algorithm must not need to know how it is orchestrated, whether it is running in parallel, in single or multithreaded mode, concurrent or not, in online or offline analysis mode.*
  JANA algorithms are ignorant of this type of information which is handled at the framework level.

- *The reconstruction framework must be open source, accessible to the entire community, and managed by a sustainable core team.*
  JLab has committed to support JANA throughout the EIC project as a full partner lab. The source is freely available from GitHub. The existing licence ties a copyright to JLab, but this will need to be revisited once contributions are made from non-JLab staff. JLab is very open to moving this forward in that regard.

Requirements

- *The reconstruction framework must be able to pass (and add) metadata and so-called slow control information to the output files, so input files are not needed and output files can stand on their own.*
  JANA allows objects of any type to be inserted into an event. Any output file writing would need to rely on tools that interface with the Data Model and so are not explicitly part of the framework itself.

- *The reconstruction framework must be able to run in streaming readout mode, that is:*
  - *with access to only parts of an event (single detector, single sector),*
  - *with events (or parts of events) appearing out of sequence,*
  - *individual algorithms must not rely on an algorithm-specific internal state to be able to make sense of disconnected parts of events.*

  JANA's Queue/Arrow architecture supports streaming at multiple levels. In particular, it can support one to many, or reordering algorithms in a natural way. The on demand design naturally supports processing of partial events. This is an extremely common exercise in GlueX.

EIC Project Detector-1
"We need a real name"

Additional assessment criteria

EPSCI
Experimental Physics Software and Computing Infrastructure

- *Amount of 'boilerplate' code that must be written by algorithm developers.*
  The ***jana-generate.py*** script generates the boilerplate code based on single or a few inputs. This includes making a complete stand-alone plugin with CMakeLists.txt file. This makes it very easy to add new components quickly.

- *Ability by the framework to avoid e.g. memory errors through interface enforcement mechanisms (e.g. const passing).*
  JANA passes pointers to const objects between factories and processors. This is required for reproducibility should the order of factory calls be changed between program invocations.

- *Ability for shared algorithm development between the two EIC detector collaborations (and/or outside of the EIC).*
  JANA factories are self contained in that they request objects and publish objects via the framework. Any detector collaboration using the same input and output classes will be portable/sharable. Furthermore, the plugin mechanism allows a plugin to provide one or more factories to any JANA executable. Thus, a single pre-compiled plugin can be used in multiple experiments.

- *Use of modern and sustainable coding practices, including in the code written by algorithm developers and other contributors.*
  JANA is maintained in a Github repository. The issues, pull requests, and release mechanisms are used to maintain the code. Automated builds and unit tests on multiple platforms are initiated by pull requests.

- *Demonstration of performance in production environments.*
  GlueX.

EIC Project Detector-1
"We need a real name"

EPSCI
Experimental Physics Software and Computing Infrastructure

Backups

The following are some notes I made a while back when trying to understand how JANA, Gaudi, and Fun4all approach the basic function of the framework. It is terribly incomplete, but may give some insight so I included it here in the backup slides.

Here I try and breakdown some example reconstruction code from ATHENA's juggler framework based on GAUDI. At the same time I try and compare this to what an equivalent JANA2 implementation would look like.

This is the first algorithm I looked at in the ATHENA repository and can be found here:

https://eicweb.phy.anl.gov/EIC/juggler/-/blob/master/JugReco/src/components/SimpleClustering.cpp

I looked at it first since the name "SimpleClustering" seemed like a good place to start.

SimpleClustering.cpp 6.21 KB

```cpp
#include <algorithm>

#include "Gaudi/Property.h"
#include "GaudiAlg/GaudiAlgorithm.h"
#include "GaudiAlg/GaudiTool.h"
#include "GaudiAlg/Transformer.h"
#include "GaudiKernel/PhysicalConstants.h"
#include "GaudiKernel/RndmGenerators.h"
#include "GaudiKernel/ToolHandle.h"

#include "DDRec/CellIDPositionConverter.h"
#include "DDRec/Surface.h"
#include "DDRec/SurfaceManager.h"

#include "JugBase/DataHandle.h"
#include "JugBase/IGeoSvc.h"
#include "JugBase/UniqueID.h"

// Event Model related classes
#include "dd4pod/CalorimeterHitCollection.h"
#include "eicd/CalorimeterHitCollection.h"
#include "eicd/ClusterCollection.h"
#include "eicd/ProtoClusterCollection.h"
#include "eicd/RawCalorimeterHitCollection.h"

using namespace Gaudi::Units;

namespace Jug::Reco {
```

This is a preamble to the file. Nothing remarkable here.

```
/** Simple clustering algorithm.
 *
 * \ingroup reco
 */
class SimpleClustering : public GaudiAlgorithm, AlgorithmIDMixin<> {
public:
```

Class is defined in implementation file in a Java-like way. This may be a stylistic choice, but definitely something allowed by GAUDI. Without a header file, the class cannot be directly used in code outside of this. Any use would have to come from properties of the class coming through one of its base classes.

```

DECLARE_COMPONENT(SimpleClustering)

} // namespace Jug::Reco
```

The class is declared to GAUDI by the DECLARE_COMPONENT *call at the bottom of the file. This is defined through a few files but eventually gets to this file and the following line:*

Gaudi/GaudiPluginService/include/Gaudi/PluginServiceV2.h

Registry::instance().add(id, { libraryName(), std::move(f), std::move(props) });

At this point I don't know if that is instantiating an object of this class or otherwise generating code that can be used to instantiate SimpleClustering objects later.

```
class SimpleClustering : public JFactoryT<Cluster> {
```

```
extern "C" {
    void InitPlugin(JApplication *app) {
        InitJANAPlugin(app);
        app->Add(new JFactoryGeneratorT<SimpleClustering>());
    }
}
```

The JANA equivalent here would be to create a class inheriting from JFactory and then report that to JANA by instantiating a JFactoryGenerator class via template.

JANA will use the JFactoryGenerator class to instantiate multiple SimpleClustering objects later.

```cpp
/** Simple clustering algorithm.
 *
 * \ingroup reco
 */
class SimpleClustering : public GaudiAlgorithm, AlgorithmIDMixin<> {
public:
  using RecHits  = eic::CalorimeterHitCollection;
  using ProtoClusters = eic::ProtoClusterCollection;
  using Clusters = eic::ClusterCollection;

  DataHandle<RecHits>       m_inputHitCollection{"inputHitCollection", Gaudi::DataHandle::Reader, this};
  DataHandle<ProtoClusters> m_outputProtoClusters{"outputProtoCluster", Gaudi::DataHandle::Writer, this};
  DataHandle<Clusters>      m_outputClusters{"outputClusterCollection", Gaudi::DataHandle::Writer, this};

  Gaudi::Property<std::string> m_mcHits{this, "mcHits", ""};

  Gaudi::Property<double>   m_minModuleEdep{this, "minModuleEdep", 5.0 * MeV};
  Gaudi::Property<double>   m_maxDistance{this, "maxDistance", 20.0 * cm};

  /// Pointer to the geometry service
  SmartIF<IGeoSvc> m_geoSvc;

  // Monte Carlo particle source identifier
  const int32_t m_kMonteCarloSource{uniqueID<int32_t>("mcparticles")};
  // Optional handle to MC hits
  std::unique_ptr<DataHandle<dd4pod::CalorimeterHitCollection>> m_inputMC;

  SimpleClustering(const std::string& name, ISvcLocator* svcLoc)
    : GaudiAlgorithm(name, svcLoc)
    , AlgorithmIDMixin<>(name, info()) {
    declareProperty("inputHitCollection", m_inputHitCollection, "");
    declareProperty("outputProtoClusterCollection", m_outputClusters, "Output proto clusters");
    declareProperty("outputClusterCollection", m_outputClusters, "Output clusters");
  }
```

Convenience declarations

Data objects in Gaudi are contained in DataHandle templated classes. It looks like these wrappers are instantiated with a pointer to the algorithm object they belong to.

Gaudi Property objects look to similarly wrap variables in a class and register it with the Gaudi system. This will allow Gaudi to know and set these values externally.

The JANA equivalent to these properties are configuration parameters. It is not clear if Gaudi expects to change these after event processing has started, but in JANA they are not expected to change. A comparable JANA call would be:

*double m_minModuleEdep = 5.0 * MeV;*
app->SetDefaultParameter("minModuleEdep", m_minModuleEdep, "...");

typo?

Input and output objects are declared explicitly in the constructor. It is not clear why this is needed in addition to the DataHandle constructors above.

```cpp
  StatusCode initialize() override
  {
    if (GaudiAlgorithm::initialize().isFailure()) {
      return StatusCode::FAILURE;
    }
    // Initialize the MC input hit collection if requested
    if (m_mcHits != "") {
      m_inputMC =
          std::make_unique<DataHandle<dd4pod::CalorimeterHitCollection>>(m_mcHits, Gaudi::DataHandle::Reader, this);
    }
    m_geoSvc = service("GeoSvc");
    if (!m_geoSvc) {
      error() << "Unable to locate Geometry Service. "
              << "Make sure you have GeoSvc and SimSvc in the right order in the configuration." << endmsg;
      return StatusCode::FAILURE;
    }
    return StatusCode::SUCCESS;
  }
```

Gaudi initialization method. This returns a value indicating if the initialization succeeds or fails.

Here, a string property of the class is used to determine if an input container should be made for MC hits.

```cpp
void SimpleClustering::Init() {
    auto app = GetApplication();

    /// Acquire any parameters
    // app->GetParameter("parameter_name", m_destination);

    /// Acquire any services
    // m_service = app->GetService<ServiceT>();

    /// Set any factory flags
    // SetFactoryFlag(JFactory_Flags_t::NOT_OBJECT_OWNER);
}
```

JANA initialization method. Unlike Gaudi, JANA does not emit a return value. In JANA, Init() is only called at event processing time if/when an algorithm is first used and so it is assumed to be required. Fatal errors in the Init() method are expected to emit errors to the logging service and to tell the application to quit via a call to app->Quit(). One may also explicitly set an exit code with app->SetExitCode(val).

```cpp
    StatusCode execute() override
    {
      // input collections
      const auto& hits = *m_inputHitCollection.get();
      // Create output collections
      auto& proto = *m_outputProtoClusters.createAndPut();
      auto& clusters = *m_outputClusters.createAndPut();
      // Optional MC data
      const dd4pod::CalorimeterHitCollection* mcHits = nullptr;
      if (m_inputMC) {
        mcHits = m_inputMC->get();
      }

      std::vector<std::pair<uint32_t, eic::ConstCalorimeterHit>> the_hits;
      std::vector<std::pair<uint32_t, eic::ConstCalorimeterHit>> remaining_hits;
```

This is the top of the execute() method which is called for every event for which the algorithm is active. The first lines are used to get the inputs for the algorithm and to create the output containers for the algorithm.

This mechanism uses the existence of a container that may or may not have been created in the init() method to determine whether to get the actual hits into the container.

JANA method that is called for every event.

```cpp
void SimpleCluster_factory::Process(const std::shared_ptr<const JEvent> &jevent){

    auto calohits = jevent->Get<DFCALHit>(); // Get input objects

    // .... Create cluster objects ....
    {
        auto cluster = new DFCALCluster( a, b, c );
        for( auto hit : myhits )cluster->AddAssociatedObject( hit );
        Insert( cluster ); //pass ownership to framework
    }
}
```

Input objects obtained as vector<const DFCALHit> calohits*

Algorithm creates cluster objects and "Inserts" them into the event using the Insert() method. One could also fill a local std::vector<> of pointers and publish those with the Set() method.

If the DFCALCluster class inherits from JObject, then the AssociatedObject mechanism can be used. This allows the framework to know about which hit objects were used to make the cluster.

Here is a comparison with Fun4All. This is taken from the following:

https://github.com/ECCE-EIC/coresoftware/blob/master/offline/packages/CaloReco/RawClusterBuilderFwd.h

I wanted to use another calorimeter clustering algorithm and this was the best I could locating with a quick search.

To start with, I should note that some of the code dealing with this is spread over a few classes:

RawClusterDefs — *Namespace. Defines RawClusterDefs::keytype*
RawCluster — *Inherits from PHObject*
RawClusterContainer — *Inherits from PHObject*
RawClusterBuilderFwd — *Inherits from SubsysReco*

```
#ifndef CALOBASE_RAWCLUSTERDEFS_H
#define CALOBASE_RAWCLUSTERDEFS_H

namespace RawClusterDefs
{
  typedef unsigned int keytype;
}

#endif
```

This is just a namespace used to define the keytype used for the RawCluster objects. Presumably this is useful for object persistence since the unique id can be reproduced if the data were replayed.

JANA has removed support for object ids in JANA2. This is due to almost never being used in JANA1. This is likely due to the heavy use of pointers which also provide unique ids within the event, but don't require lookup tables to get at the object data.

```
class RawCluster : public PHObject
{
 public:
  typedef std::map<RawTowerDefs::keytype, float> TowerMap;
  typedef TowerMap::iterator TowerIterator;
  typedef TowerMap::const_iterator TowerConstIterator;
  typedef std::pair<TowerIterator, TowerIterator> TowerRange;
  typedef std::pair<TowerConstIterator, TowerConstIterator> TowerConstRange;

  ~RawCluster() override {}
  void Reset() override { PHOOL_VIRTUAL_WARNING; }

  PHObject* CloneMe() const override { return nullptr; }

  int isValid() const override
  {
    PHOOL_VIRTUAL_WARNING;
    return 0;
  }
  void identify(std::ostream& /*os*/ = std::cout) const override { PHOOL_VIRTUAL_WARNING; }
  /** @defgroup getters
   *  @{
   */
  //! cluster ID
  virtual RawClusterDefs::keytype get_id() const
  {
    PHOOL_VIRTUAL_WARN("get_id()");
    return 0;
  }
  //! total energy
  virtual float get_energy() const
  {
    PHOOL_VIRTUAL_WARN("get_energy()");
    return NAN;
  }
```

```cpp
class RawClusterContainer : public PHObject
{
 public:
  typedef std::map<RawClusterDefs::keytype, RawCluster *> Map;
  typedef Map::iterator Iterator;
  typedef Map::const_iterator ConstIterator;
  typedef std::pair<Iterator, Iterator> Range;
  typedef std::pair<ConstIterator, ConstIterator> ConstRange;

  RawClusterContainer() {}
  ~RawClusterContainer() override {}

  void Reset() override;
  int isValid() const override;
  void identify(std::ostream &os = std::cout) const override;

  ConstIterator AddCluster(RawCluster *clus);

  RawCluster *getCluster(const RawClusterDefs::keytype id);
  const RawCluster *getCluster(const RawClusterDefs::keytype id) const;

  //! return all clusters
  ConstRange getClusters(void) const;
  Range getClusters(void);
  const Map &getClustersMap() const { return _clusters; }
  Map &getClustersMap() { return _clusters; }

  unsigned int size() const { return _clusters.size(); }
  double getTotalEdep() const;

 protected:
  Map _clusters;
```

The RawClusterContainer class is interesting because it really serves as a customized container class for RawCluster objects. It has several methods like AddCluster, getCluster, getClusters, … that include the word "cluster" in their names. These do not seem to be doing anything special that any other container class would not already be doing. It is unclear why a more general (templated) container class is not used which could provide more uniformity in the code.

n.b. getTotalEdep() looks to be the only method that has functionality that would not be provided by a generic container class.

In JANA, the JFactory (i.e. algorithm) class that produces the data objects owns them and serves the combined purpose of the RawClusterContainer and RawClusterBuilderFwd classes. The JFactory class is actually a template itself where the template parameter is the specific type of primary data object the factory produces.
n.b. More than one object type can be produced by a JFactory. The supplementary types would use Insert() to add them to the event and would no longer be owned by the factory. This would make no difference to the end user. The emphasis on having a factory produce a single, primary object type is meant to encourage modularity in the overall design by having more, smaller algorithms.