# Opening up the Black Box
# Inside the NuWa Framework

Brett Viren

Physics Department




UW DB Ana WS

# Outline

Framework vs. Other Paradigms

Gaudi Fundamentals

NuWa Layers

C++ vs. Python

Links to More Information

# Design Paradigms or What is a Framework?

Some possible software design strategies with analogies:

- Ad-hoc: No design, everyone writes as they wish.
  - Leads to "code fiefdoms" connected by non-standard file formats.
  - Like each person building room in a house starting from trees.
- Toolkits : major functionality in libraries shared by all.
  - Tends to inconsistent, duplicated code, data model and file formats.
  - Each room built starting with common prefab walls and pipes.
- Frameworks : toolkits + shared main program, data, files.
  - Users focus on plugging in just the specific feature they want.
  - Moving in to an architected and finished house and just worrying about where to put the furniture.

Frameworks move as much common functionality out of user code as possible. They solve problems in general and provide well defined entries for users to add targeted, unique contributions. This removes some user "freedom" but provides structure and assurances.

# A High Level View of Gaudi

- The basis of the Gaudi framework is a set of cooperating roles.
- The roles are defined with an intent to process (or generate) data and are formalized as a set of abstract base classes.
- Concrete implementation then fulfills the role in a manner specific to an experiment's needs.

To understand Gaudi is to understand the roles of the actors involved.

# Interfaces Classes

Roles are "implemented" as Interface classes.

- Have names beginning in "I" by convention.
- Define pure-virtual methods to express the role.
- Have (almost) no implementation.
- Have a unique ID #.

At the lowest level, Gaudi only deals with Interfaces.

# Interface Example, IProperty

Properties are key, value pairs. They provide for user configuration. The Interface to implement is IProperty:

```
class IProperty : virtual public IInterface  {
public:
  // Reimplemented by any Interface subclass.
  static const InterfaceID& interfaceID();

  // Sub classes must provide:
  virtual StatusCode setProperty(const Property& p) = 0;
  virtual StatusCode setProperty(const string& s) = 0;
  virtual StatusCode setProperty(const string& n,
                                 const string& v) = 0;
  virtual StatusCode getProperty(Property* p) const = 0;
  virtual const Property& getProperty(const string& name) const
  virtual const Property& getProperty(const string& name) const
  virtual StatusCode getProperty(const string& n,
                                 string& v) const = 0;
  virtual const std::vector<Property*>& getProperties() const =
}
```

# Component Classes

- While Interfaces are abstract, Components are concrete objects based on one or more Interfaces.
- You can think of a Component as an object that can display one or more "facets", each of which is an Interface.
- Gaudi provides various Components which implement their Interfaces either fully or partially but with default "dummy" implementations for the rest.
- These Components are either usable as-is or meant to be subclassed to provide full functionality.

# Basic Component: Algorithm

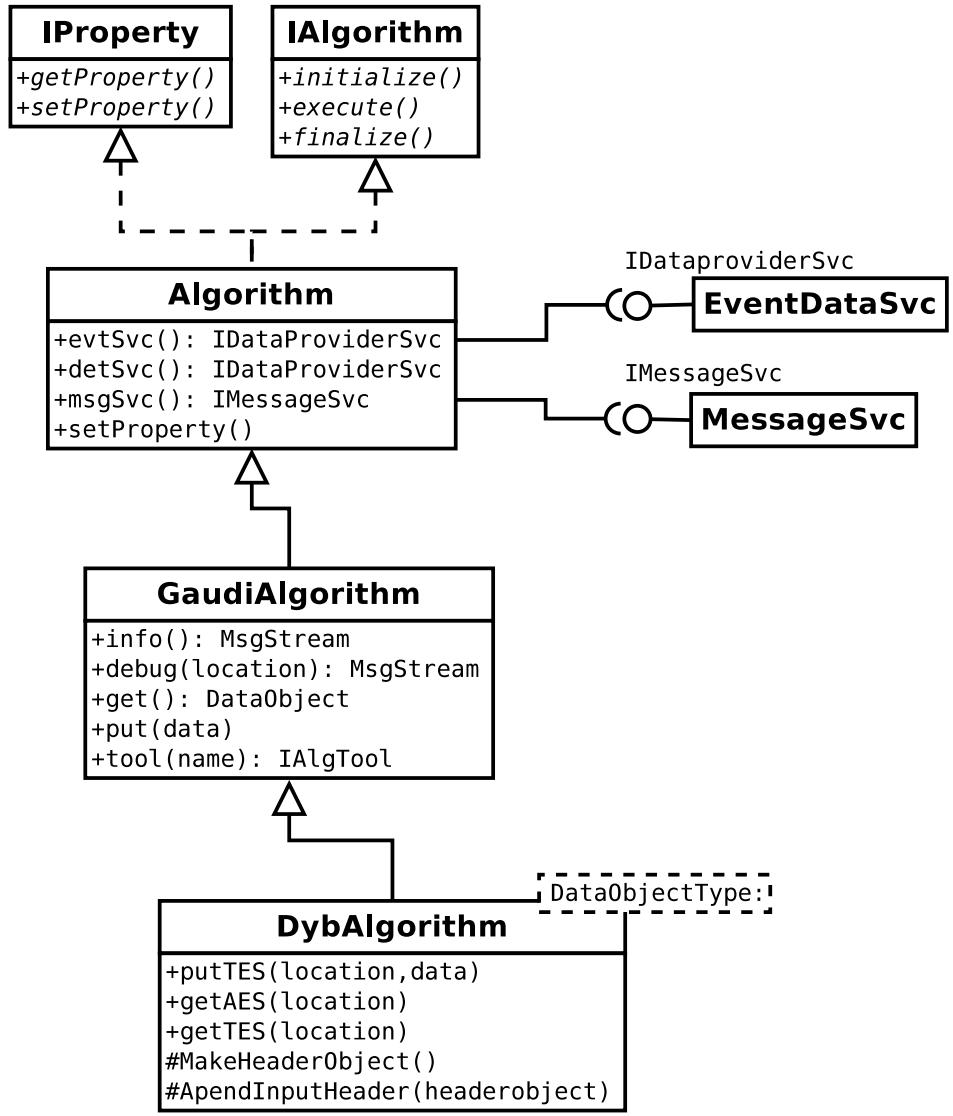
Modular units of processing.

- Entries into user code:

  `initialize()` once at beginning
  `execute()` once per "event"
  `finalize()` once at end

- `Algorithm` gives access to services.
- `GaudiAlgorithm` adds syntactic sugar to access services.
- `DybAlgorithm` handles `inputHeaders` and AES access, Use if you produce output data.

Algorithms are independent from one another but can be chained together in one job.



# Basic Components: Tools and Services

Two similar Components:

Tools:

- **Provide reusable procedures.**
- Shared or private copies.
- Mostly used by Algorithms.
- Base component class: `AlgTool`
- Eg: GenTools' `GtPositionerTool`

Services:

- **Access to shared resources.**
- Shared singleton instance.
- Used by most objects.
- Base component class: `Service`
- Eg: DetHelpers' `PmtGeomInfoSvc`

Both:

- Implemented by subclassing their respective base component plus an Interface defining their specific methods.
- User configurable.

Note: a good design pattern is to use Algorithm to marshal data and, put all the "smarts" in Tools.

# Some Other Basic Components

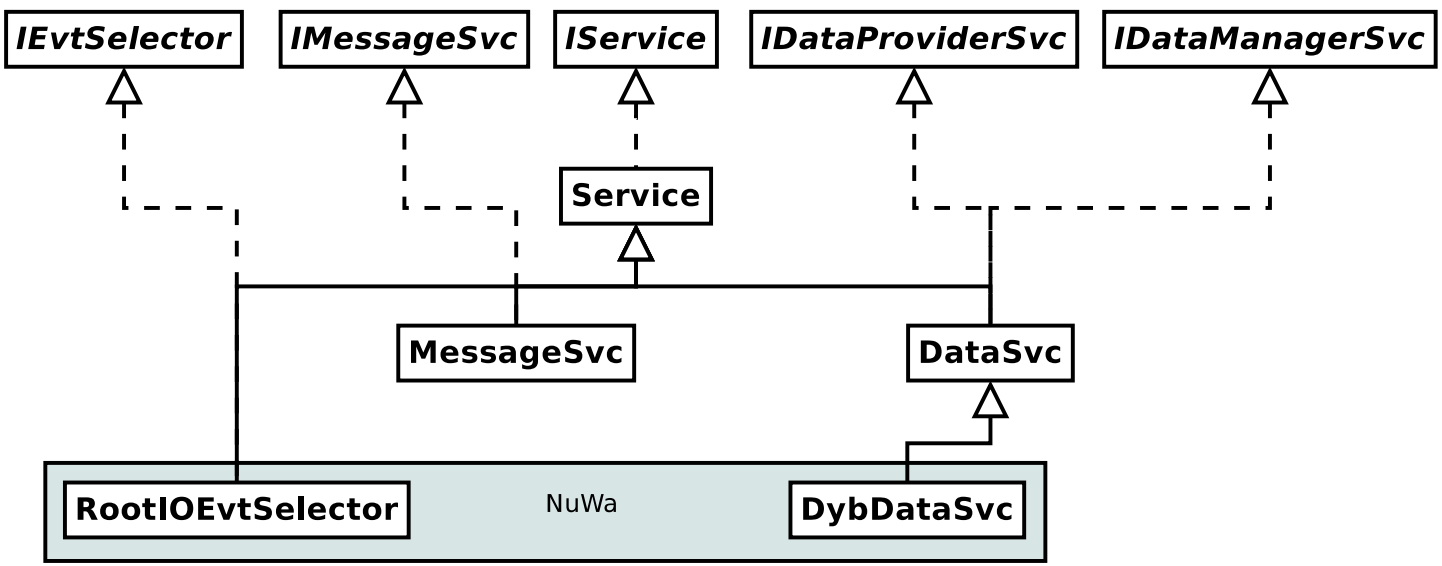
Other high profile, basic Components:

- Converter used to translate between objects in memory and files.

Event Selector defines what "next event" means.

Data Service an in memory store of data objects.

Message Service logging.



# NuWa Services

Some services specific to Daya Bay:

**Archive Event Store (AES)** :
A secondary Data Service that provides a way to look back in time to form delayed coincidences.

**PMT Geom Info** :
Locate PMT geometry by PMT ID# or name.

**Coordinate System** :
Find applicable coordinate system given a global point.

**Cable Mapping** :
Convert between sensors (PMT/RPC) and electronic channels.

**Statistics** :
Improved histogram/ntuple/TObject handling.

# Access Services

In general, services must be looked up via their desired interface and componenet class name before use. This should be done once in `initialize()`. Using these methods, they will be released automatically at `finalize()`.

In C++, using at least `GaudiAlgorithm` as base:

```
m_pmtgeo = svc<IPmtGeomInfoSvc>("PmtGeomInfoSvc");
```

In Python, using at least GaudiAlgo as base:

```
self.cabling = self.svc('ICableSvc','StaticCableSvc')
```

# Wrapped Services

Many services are wrapped with "syntactic sugar":

MessageSvc:

```
// C++ using GaudiAlgorithm base
warning() << "Oh no!  The printer is on fire" << endreq;

# Python using DybPythonAlg as base
self.info('This is an %s parrot'%state_of_parrot)
```

Event Data Services:

```
// C++
SimHeader* sh = 0;
sh = get<SimHeader>("/Event/Sim/SimHeader");
put(sh,"/Event/Sim/SimHeader");

# Python
sh = self.get('/Event/Sim/SimHeader')
```

# NuWa Jobs

- Single main program: `nuwa.py`
  - Rich command line options control I/O, logging, basic configuration.
- Framework defaults configured automatically.
- User configuration through Python modules and scripts.
  - Most packages provide sane default configuration
  - User and NuWa modules can take own command line options
- Python or C++ Algorithms supported.

# Life Cycle of a NuWa Job

A job go through several states (state names are my terms):

**Configure** state progresses through three steps:[1]

1. all job modules on command line are loaded
2. all `configure()` methods are called
3. all `run()` methods are called

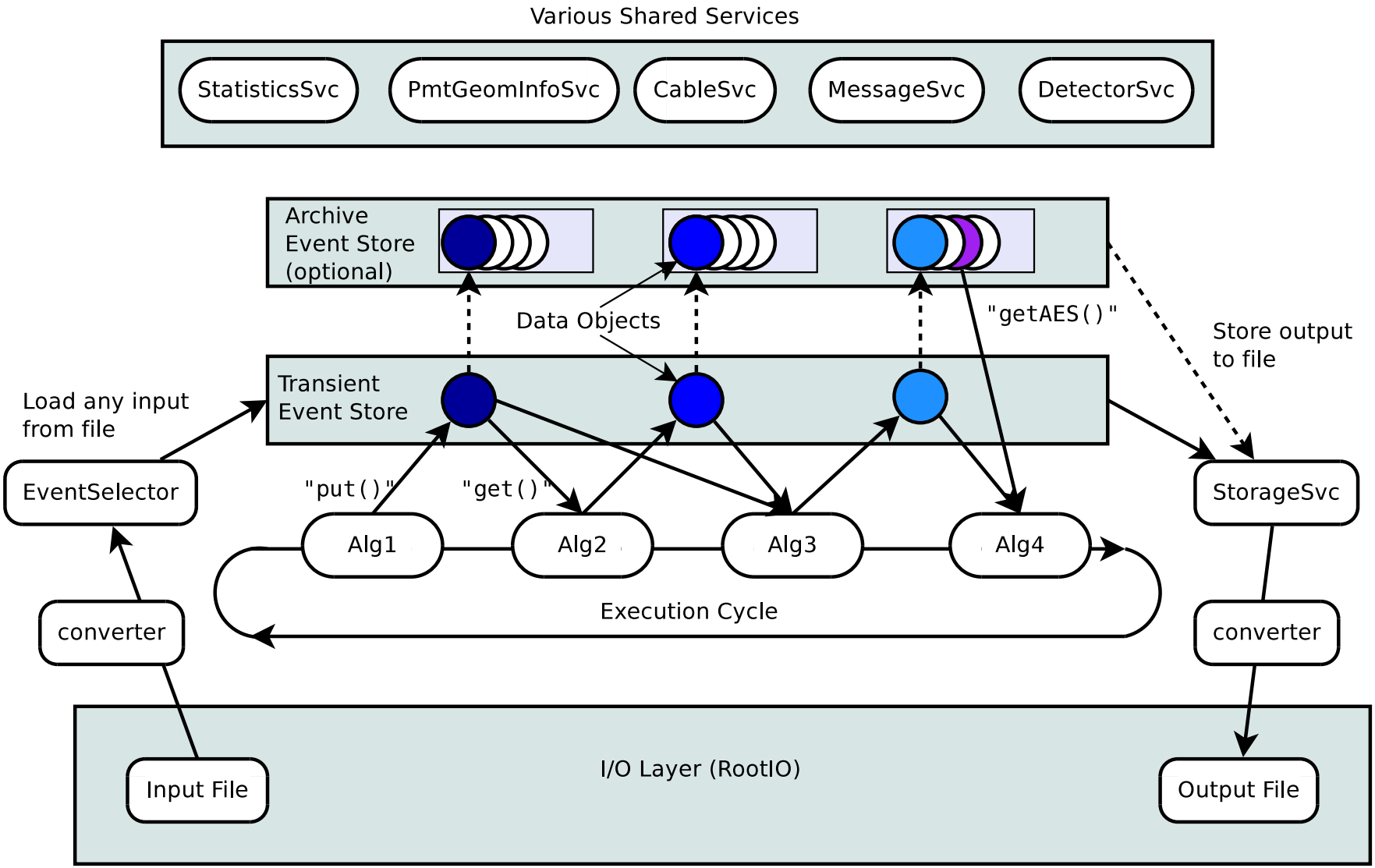
**Initialize** Algorithms' `initialize()` methods are called.

**Executing** Algorithms' `execute()` methods are called on each *execution cycle* (ie, "event")

**Finalize** Algorithms' `finalize()` methods are called, services shutdown and job ends.

[1] More on this below.

# Cartoon of a Running Job



# Symmetries and Asymmetries between C++ and Python

Symmetries:

- Implement Algorithms
- Access the Data Model objects and Gaudi components via interfaces
- Use of C++ classes wrapped for Python (includes all of ROOT)

Asymmetries:

- All Python algs run after all C++ algs each execution cycle (see next)
- Most user Python is "configure time" code, most user C++ is "run time" code (also see next slide).
- Python can access (some) C++ objects but not vice versa
- Some C++ syntax has no Python equivalent
  - Iterators may be able to use `DybPython.Util.irange`
  - No way to create "new" instances of objects or STL collections
- Python algs may be significantly slower than equivalent C++ algs

# Some Potential Python Confusions

In the context of NuWa, there are two types of Python code:

**Configuration** code runs before Gaudi starts up.

- First, top-level code in all modules is run as the module is loaded in order set by the command line list of modules.
- Next, all existing "`configure()`" methods are called in the same order.
- This code can:
  - Instantiate and set Gaudi *Configurables* - classes that hold the Properties of Components.
  - Define any Python Algorithm classes.

**Run-time** code runs just after Gaudi starts up

- All existing "`run(app)`" methods are called in same order as above.
- In this method any Python algorithms should be instantiated and added to the job via "`app.addAlgorithm(alg)`".

# More Information At

- Documentation Portal in the wiki.
- DocDB 2941 tutorial on running and configuring jobs.
- `dybgaudi/DybExamples` and the tutorial project gives (usually) working examples.