Fermilab

Managed by Fermi Research Alliance, LLC for the U.S. Department of Energy Office of Science

Framework design experience from *art*

Marc Paterno

EIC software workshop

25 September 2015

What is a framework?

- From *Wikipedia*:

  "… a **software framework** is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software"
- The "generic functionality" provided by *art* is a *command-line-driven event-processing framework application*.
  - *command-line-driven*: the application is not interactive
  - *event-processing*: the program processes a sequence of events, as specified by the user
- *User-written code*, in this case, is provided by the collaborators on experiments using *art*.
- Importantly, the framework is part of a larger "ecosystem".

What are the parts of the "ecosystem"

- Source code under version control (we use git)
- A build system: *art* provides one, but does not require its use
- Release, dependency, and environment control
  - strict control over library versions: binary compatibility is guaranteed
  - *art* relies upon a system called UPS, mostly behind the scenes
  - environment variables used to control PATH, dynamic loading of libraries
- Umbrella packages to guarantee binary compatibility
- *art*: the framework itself
- Supporting products (configuration, message logging, etc.)
- Multiple package distribution options:
  - a web-based package distribution system (scisoft.fnal.gov)
  - Use of CVMFS, especially for grid use
- Available connection to data handling (decoupling is important)
- Curated tools that work together: ROOT, Geant4, python, numpy, Boost, etc. Binary compatibility guaranteed if you use our builds.

What a framework gives you

Allows you to write your physics code without worrying about the infrastructure.
Makes it easy to work with others.
<u>But not for free – you have to learn it!</u>

Some people find such a system <u>constraining</u>:

- **Infrastructure is hidden behind the scenes from you**
- **Your ideas may not be included**
- **You have to trust a system you didn't write**
- **You miss out on the fun of writing super-cool complicated C++ code**

Some people find such a system <u>liberating</u>:

- **You can concentrate on physics code**
- **Your C++ is pretty easy (you are *using* a complicated system, not *writing* it)**
- **You get to miss out having to maintain the complicated C++ code (yay!)**
- **You can use code from others and share yours with others**
- **You can get services for free (e.g. data handling)**

In g2migtrace/src/primaryConstruction.cc

```
// constructionMaterials is essentially a "materials library" class.
// Passing to to construction functions allows access to all materials

  /**** BEGIN CONSTRUCTION PROCESS  ****/

  // Construct the world volume
  labPTR = lab -> ConstructLab();
  // Construct the "holders" of the actual physical objects
#ifdef TESTBEAM
  ArcH.push_back(labPTR);
#else
  ArcH = arc->ConstructArcs(labPTR);
#endif
  // Build the calorimeters
  //  cal -> ConstructCalorimeters(ArcH);
   station->ConstructStations(ArcH);
#ifndef TESTBEAM
  // Build the physical vacuum chambers and the vacuum itself
  VacH = vC -> ConstructVacChamber(ArcH);
```

What if we have a different test beam?

What if I want a different detector configuration?

this kind of code is hard to excise later

I don't think we can't simultaneously maintain this code and our sanity

What are some of the event-processing tasks?

1. Simulation of detector response to events
2. Reconstruction of real or simulated events
3. Calibration studies
4. Analysis: making plots (or at least histograms and such)!

- All of these tasks can be performed in the same framework.
- All the modules you may write can be re-used in any relevant event-processing context.

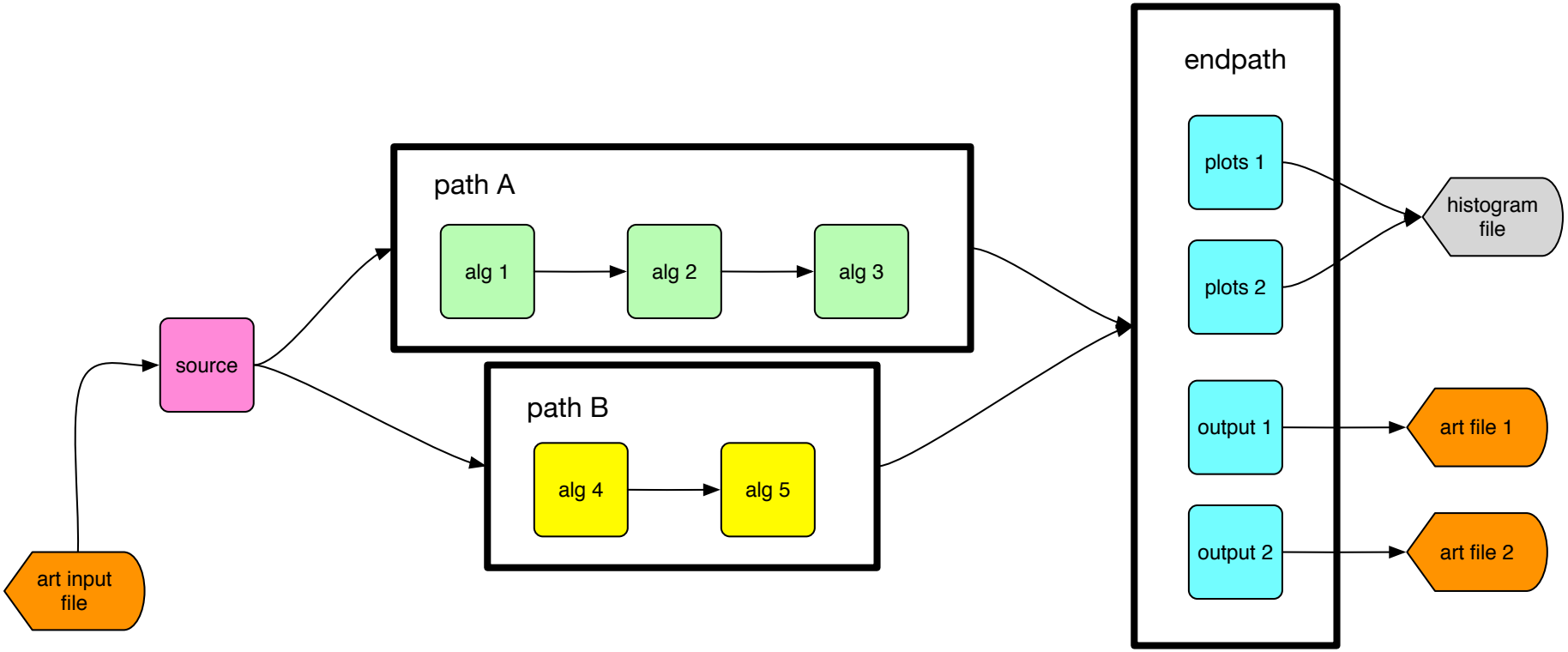
The genesis of *art*

- *art* is a fork of the CMSSW framework, the framework used by the CMS experiment at the LHC.
  - Many in the initial *art* team were also designers of CMSSW
  - The fork was done in 2009; then only considered by Mu2e
  - Simplified and made suitable for multiple experiments in 2010
- We replaced
  - the build system
  - the packaging system that allowed easy use of external products
  - plugin management, to simplify user-defined data product generation
  - the configuration language (replaced Python with JSON-ish FHiCL)
- We added the ability to ship a release to be used by experiments as an “external product”

What does the framework program do for you?

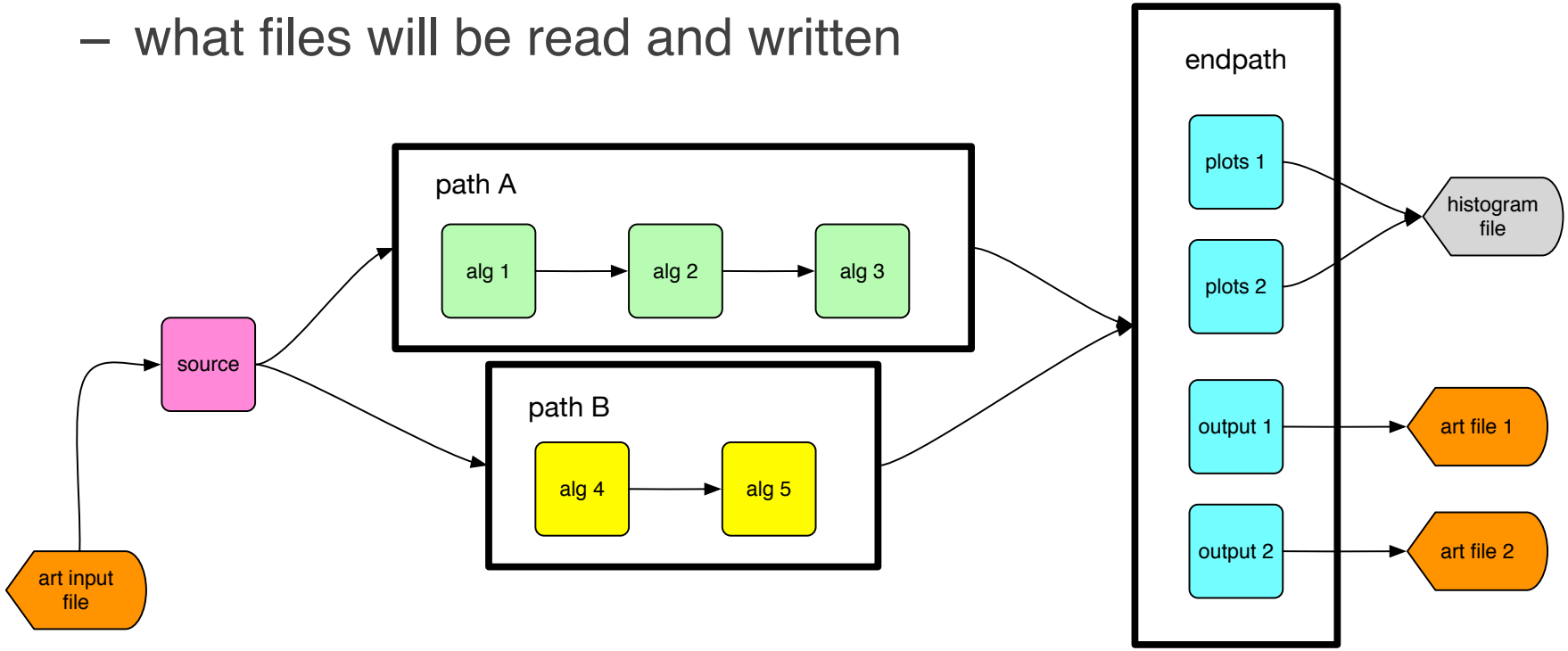
- Mostly the framework exists to handle the tasks in event processing that you don't care much about, but which have to work
  - reading input, writing output
  - loading and configuring the plugin modules you want to run
  - keeping track of how outputs were generated ("provenance tracking"); critical for reproducibility
  - organizing histogram output
  - Services to manage access to "global resources": geometry information, calibrations, …
  - systematizing the handling of error conditions (exception classes and a pattern for their use)
  - timing modules, measuring memory use, tracking execution, …
- The framework does *not* know about physics

A high-level view of a configured *art* program



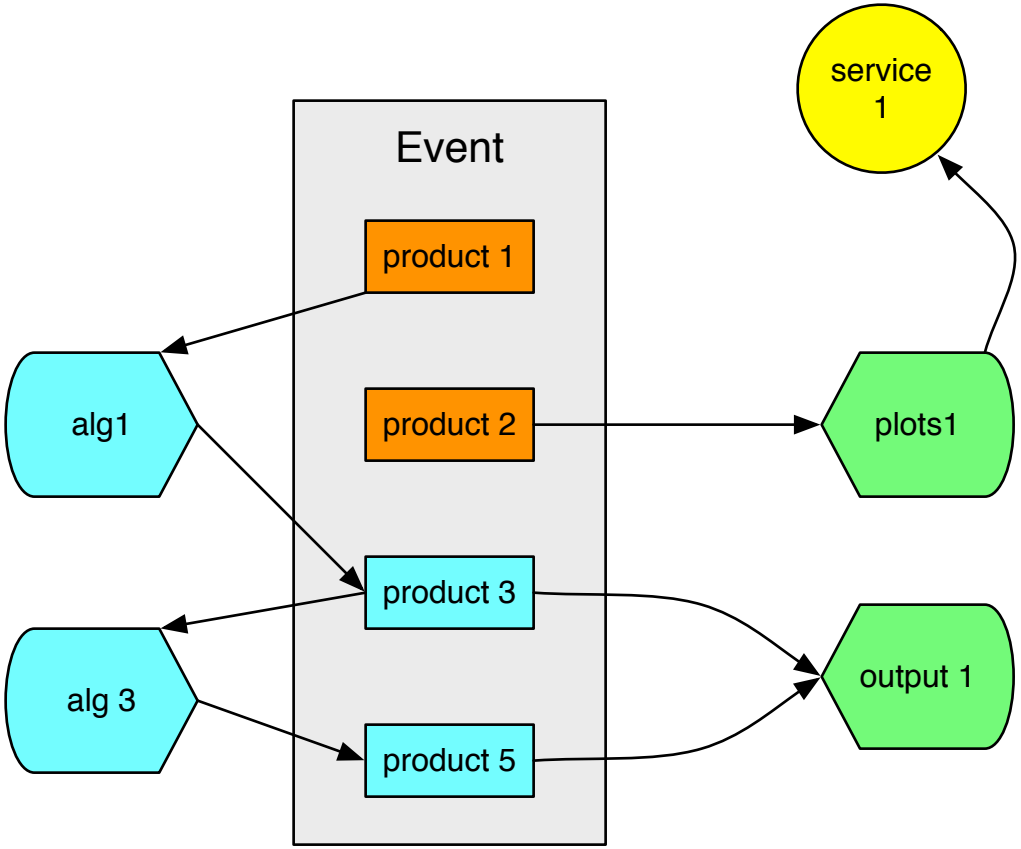
Choosing algorithms to run

- Algorithms (simulation, reconstruction, or just analysis code) is built into classes, put into dynamic libraries called *modules*.
- Text files (in a language called FHiCL) declare
  - what modules will be loaded, and in what order they are to run
  - what files will be read and written



Accessing data

- Modules *never* communicate with (call) other modules.
- Modules can call *services* (e.g., to create histograms managed by ROOT).
- Mostly, modules interact with an *Event*.
- An *Event* is just an organized collection of data products, with information about them (metadata).

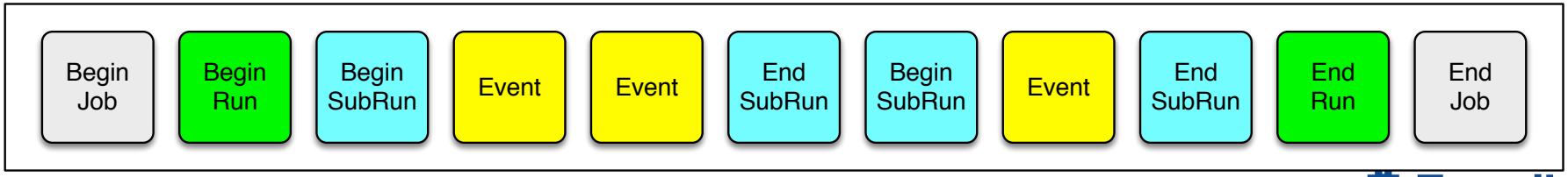


Data: events, subruns, runs, data products

- An *Event* is the "atomic unit" for data processing, and is like a in-memory database of user-defined data products
  - modules are passed a whole event, pick out the parts they want
  - producers and filters can put new data products into an event
  - *art* provides facilities for creating data product classes, but doesn't actually contain any such classes. Your experiments define them.
- A *SubRun* is:
  - a sequence of events, collected or simulated under some consistent running conditions
  - an event-like container for subrun products
- A *Run* is like a subrun, only bigger.
- The rules for defining subruns and runs belong to your experiment, and are not part of *art*.
- Events labeled with an *EventID*, which contains a triplet of run number, subrun number, and event number.

Phases of processing: callbacks and the module API

- Modules are classes, so have constructors and destructors.
  - do as much initialization as possible in the constructor
- Modules have member functions to handle the event loop
  - **begin/end job**: initialization not possible in the constructor can be done here; should be undone at end job. Called before files are open.
  - **begin/end run**: called when a new run is encountered in a file (some subtleties ignored for now)
  - **begin/end subrun**: similar to above, but for subruns
  - **event**: this is the main processing function for most modules
- Some module types can read from and write to the event; some can only read from the event.



5 different module types

- *Sources* are the things that provide the sequence of runs / subruns / events to be processed. art provides a few widely-used sources and tools to write your own.
- *Producers* and *filters* create new data products; filters also return a status that can terminate a path.
- *Analyzers* can't create new products, but can write other output (e.g. histogram files, ntuples).
- *Output modules* write output files.
- Exactly one source, and any number of the others types of module, can be used in the same program.
- Multiple instances of the same type of module are allowed

The difference between a module *type* and *instance*

- A module *type* is also a C++ *type*, that is, a *class*.
- One can have multiple instances of the same data type, with distinct identity and state:

```
std::string greeting { “hello” };
std::string farewell { “goodbye” };
```

- Similarly, a framework program can have multiple instances of the same module type:
  - Several instances of the same tracking algorithm, each with different values of some configurable parameters.
  - Several instances of *RootOutput*, each writing its own output *art*-ROOT data file.

Services

- Services provide access to program-wide information or facilities.
- Service can be access (almost) anywhere, at (almost) any time
  - can be used in module constructors
- *art* provides some services
  - examples include timing of modules, controlled creation of ROOT histograms
- Experiments also write services
  - Some are provided by LArSoft to many experiments
  - Some are completely experiment-specific
  - examples include access to geometry information, and calibration information

Maintenance and support tools & efforts

- Meetings
  - *art* stakeholders: weekly meeting with representatives of experiments
  - triage meetings: weekly team meeting to schedule work
- Mailing lists
  - art-users@fnal.gov (open to all)
  - artists@fnal.gov (sends mail to the core developers)
  - each experiment will have one or more lists
- Issues (feature requests, bug reports) handled at Redmine site
  - anyone can report a suspected bug
  - we ask for help in getting the report into the right tracker
    - experiment code bugs in experiment's bug tracker
    - infrastructure bugs in art issue tracker
  - we ask people to discuss feature requests within their experiment, or on the art-users list, before submitting a feature request
- Project management done using the Redmine site
  - prioritize work based on experiment needs
  - the tool isn't great, but it is usable (and it is what we have)

Integration into workflows

- The FHiCL configuration language is designed to make it easy to provide automated modifications without parsing
  - e.g. appending a new line to change any configuration parameter
- Runtime environment and delivery system is designed to allow integration of workflow system tools
  - e.g. data-set handling (SAM) introduced by a 3rd-party UPS product which depends on *art* and SAM; *art* does not depend upon it.
- Delivery of software to grid sites made easy with CVMFS.
  - UPS product trees can be made available anywhere

Original design features (mostly) not now in *art*

- Path specification design provided consistency checking at program start-up.
- Modules declared what data they read, as well as what they created; enables verification of correctness, and computing of an efficient processing graph.
- Multithreaded design
  - does not improve throughput, but decreases memory use
  - this has now been added to CMSSW, using a task-based model
  - not yet in art; need is less.
- Use of an internal database (SQLite) to store metadata
  - we have recently added this, but don't use it to its full potential yet

What would I do differently today?

- Plan for concurrency from day 1.
  - We should not have allowed ourselves to be argued out of this.
- Plan for use of HPC-style resources
  - varied architectures
  - many cores, each with limited memory; think of MIC
  - rely on fast networking, avoid use of files as much as possible (we do this with the *artdaq* DAQ toolkit): distributed programming
- Plan for polyglot programming
  - allow use of other languages where possible; e.g. we are soon to deploy support for writing analysis modules in Python.
- Enforce HPC-ready data format
  - structure-of-arrays, rather than array-of-structures
  - simple data, with sophistication added by “wrappers”, classes that provide functions to go with the simple data of the data product
- Keep I/O technology choices open.
- Make everything open-source from day 1.

Things to think about very hard

- Can you share effort with someone else?
  - The neutrino and muon program experiments at FNAL have saved a great deal of effort by not inventing multiple frameworks. CDF and D0 (Tevatron) did not do this; CMS and ATLAS (LHC) did not do this.
- How will you teach your users?
  - Most will not be software experts.
  - Many will not care about software quality: they have a job to do today, and no time to worry about the future. Framework supporters have to worry about the future!
  - People who care little about quality can produces reams of code quickly, and so most examples of use might be bad examples.
  - Tutorial documentation, and task-based documentation, is very hard to produce – and is of great value.
- How will you get feedback from your users?

A few final observations concerning language

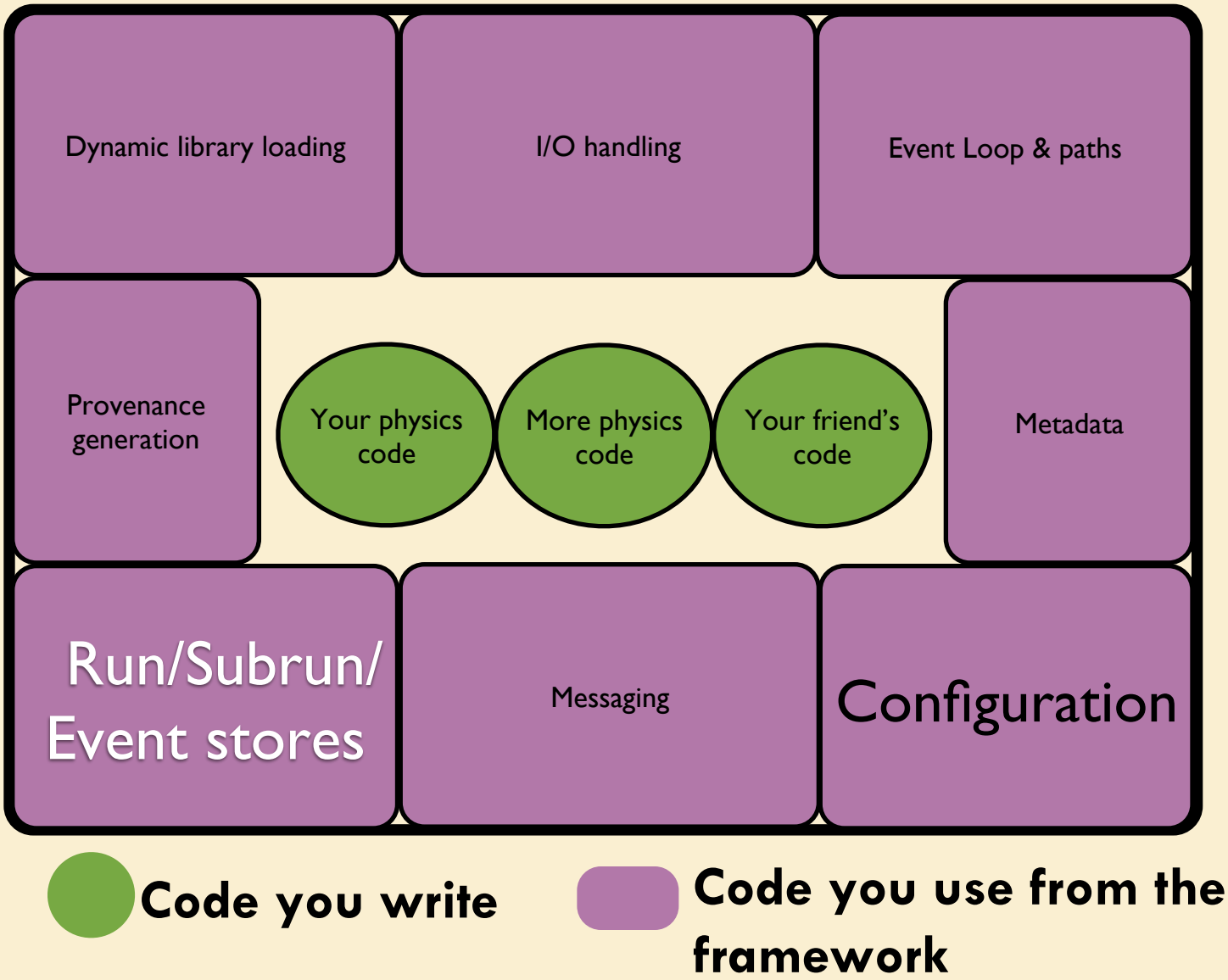
- What language should you use?
  - I think you should choose C++, but you should choose it by intention, not by accident. Know why you do not choose some other.
- Keep up with the language standard.
  - Experimenters are naturally resistant to change
  - But languages evolve for good reason
  - e.g. C++ move semantics, *shared_ptr* and *unique_ptr*, variadic templates.
  - e.g. using template metaprogramming to help avoid user errors, and to make library use simpler (we couldn't survive without SFINAE)

Summary

- Software design is mostly a “people thing”
  - coding is not so hard
  - deciding what you want the code to do is much harder
  - you can err both by too much planning (“analysis paralysis”) and by too little planning (this is very common in HEP).
- Design failures are usually *analysis* failures: not enough thought given to all the necessary cases.
- Flexibility is key:
  - you probably can’t “design one to throw away”
  - you probably can start with the most important features, in a design where extensibility is planned for.

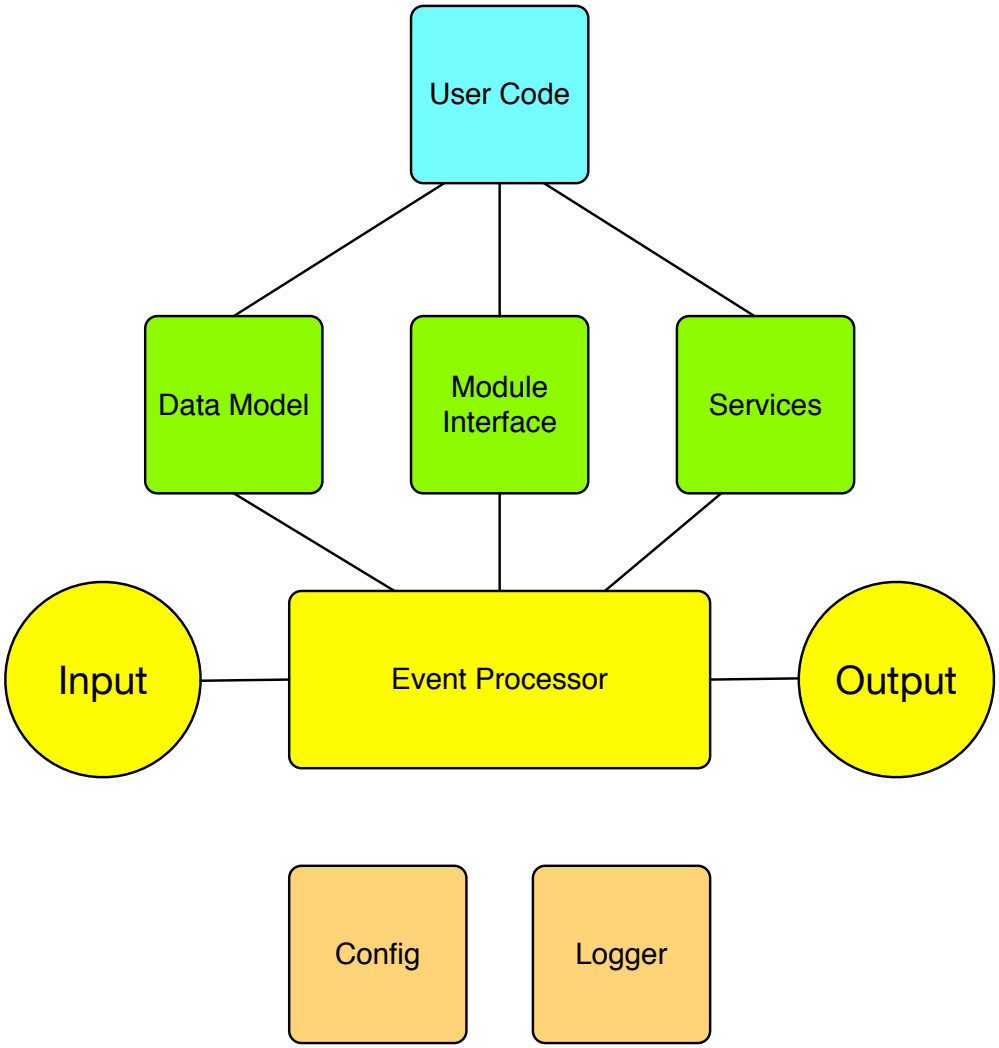
Extra slides

What does a framework do?



What are the parts of the *art* framework?

- User code is what experimenters provide.
- Services provide access to global facilities.
- Data model provides the representation of event data.
- Event processor is the "event loop", the core of the framework.
- Configuration and logger systems can be used by everything.



What might a program look like without a framework?

- Data products are read from input file.
- New data products are created by algorithms.
- Plots are created and written out.
- Data products are written to several output files.
- We want to be able to improve any algorithm without breaking others. We want *loose coupling*.

```
// pseudocode! not real C++.
// Part of the body of main
read(infile, &prod1, &prod2);
alg_1(prod1, &prod3);
alg_2(prod2, &prod4);
alg_3(prod3, &prod5);
plots1(prod2, plotfile);
plots2(prod3, prod4,
       plotfile);
write(outfile1,
      prod3, prod5);
write(outfile2,
      prod2, prod4);
```

Loose coupling *vs.* tight coupling

- Algorithms that are interwoven are hard to modify
  - changes in one part of the code often break code elsewhere
  - programs that are hard to modify are hard to improve and hard to extend with your own ideas
  - interwoven = tight coupling
- Loose coupling increases flexibility
  - replace algorithms you don't like with ones you do
  - extend data structures without breaking old code
  - don't need to "rebuild the world" because of local modifications
- Loose coupling can be applied at every level
  - between classes
  - between libraries
  - between sets of libraries (packages)
  - this has influenced the design of *art* at every level.

Where does your code go?

- Of course, all code goes into a source code repository!
- You only need to have the source code you are modifying
  - You are not modifying *art* itself
  - You may be modifying experiment code, or LArSoft code
- Your experiment many have many packages.
- The organization of your experiment's code determines how much (or how little) code you need to have access to.
- To make builds fast, it is best to check out only what you have to, and to use pre-built libraries as much as you can.
  - *art*, ROOT, Geant4, boost, … many large libraries are provided pre-built for you.
  - If you are *using* LArSoft (as opposed to *modifying* it), you can use the pre-built libraries.

Getting input

- *Sources* are the things that tell the framework what *runs*, *subruns*, and *events* are to be processed.
- Some sources read data files (e.g. *RootInput*, which reads the *art*-ROOT data file format, as written by *RootOutput*).
- One source (*EmptyEvent*) creates events containing no products; it is widely used in simulations.
- Experiment often have specialized inputs:
  - to read file formats (e.g. written by your DAQ system); these will have specialized sources created to read them;
  - to read from a live DAQ system
  - to do specialized manipulations of data from the file, before it is given to the framework

Making plots (and other analysis tasks)

- Not all algorithms have to do with simulation or reconstruction tasks.
- Not all algorithms create new data products for other algorithms.
- Some algorithms accumulate statistics about event data
  - calculate statistical summaries for printing
  - mostly, create and fill histograms (or other types of plots)
- The framework provides a module variety called an *analyzer* for such tasks.