Fermilab

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

Framework introduction

Marc Paterno

art/LArSoft course

3 August 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 you and your colleagues.
- Importantly, the framework is part of a larger "ecosystem".

Why do we have a framework (and other supporting stuff)?

- To make it easier to work together.
- Why is that important?

Science demands reproducibility

- **results must come from official code**
- **must be able to share that code**
- **a framework provides the environment through which code can be shared**

- To make it easier to do your own work
  - "We just want to make plots!" –Adam Lyon
  - The framework does the parts of event processing you don't care about, but just want to work.

What are some of the event-processing tasks?

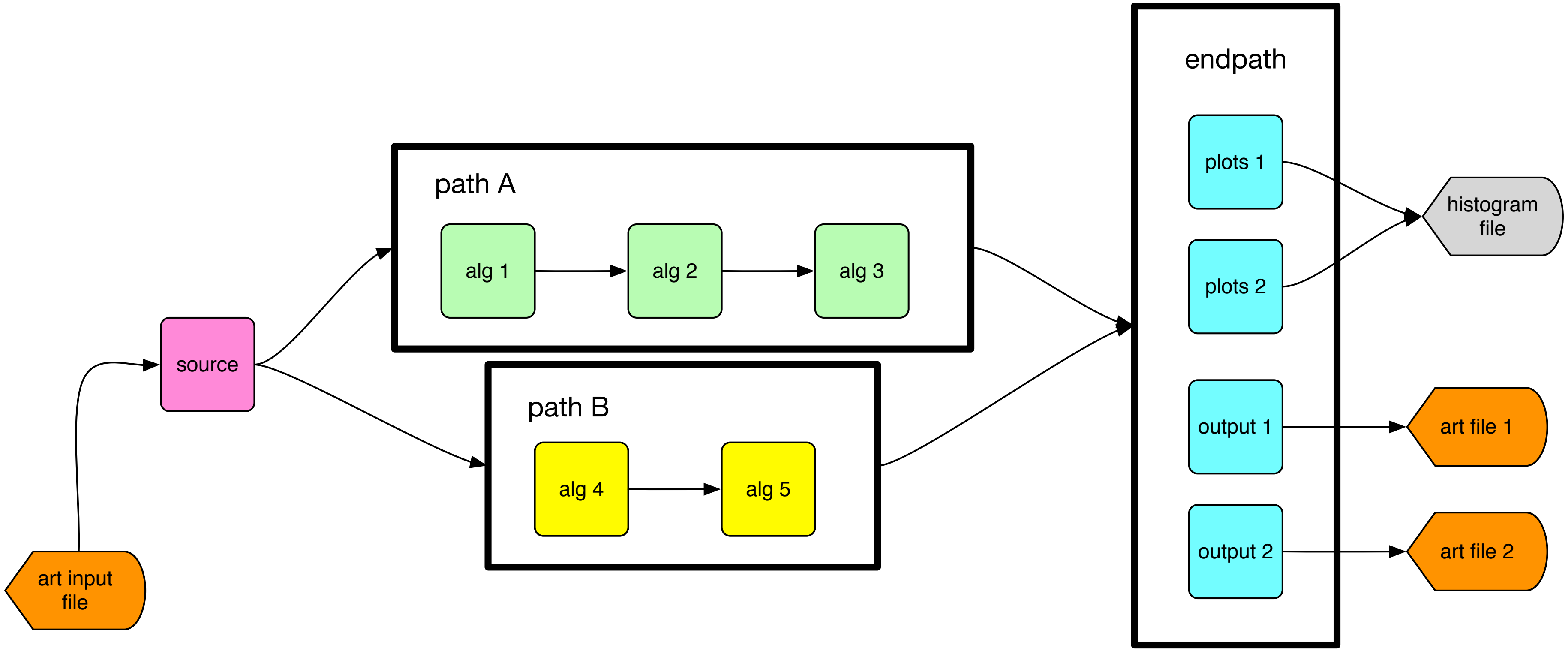
1. Simulation of detector response to events
2. Reconstruction of real or simulated events
3. Calibration studies
4. Analysis: making plots!

- 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.

What does the framework 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
  - loading algorithms you want to run
  - configuring those algorithms
  - writing output
  - keeping track of how outputs were generated ("provenance tracking"); critical for reproducibility
  - organizing histogram output
  - access to "global resources": geometry information, calibrations, …
  - systematizing the handling of errors
  - timing modules, measuring memory use, tracking execution, …
- The framework does *not* know about physics
  - You get to do the fun part

What does the framework program look like?



What are the parts of the "ecosystem"

- Source code under version control
  - Your experiment uses one, you'll have to learn about it (but not here)
  - Different experiments use different tools
    - git (many)
    - subversion (fewer)
- A build system
  - Your experiment has one, you'll have to learn about it (but not here)
  - For this class, you'll continue using *cetbuildtools* (for LArSoft users, you'll be introduced to *mrb* on Friday)
- Release, dependency, and environment control
  - *art* relies upon UPS, mostly behind the scenes
  - environment variables used to control PATH, dynamic loading of libraries, etc.
  - You'll see a little of this here.
- *art*: this is what we'll be learning about.
  - the framework itself
  - supporting products, *e.g.* configuration language, messaging, *etc*.

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*.

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*.

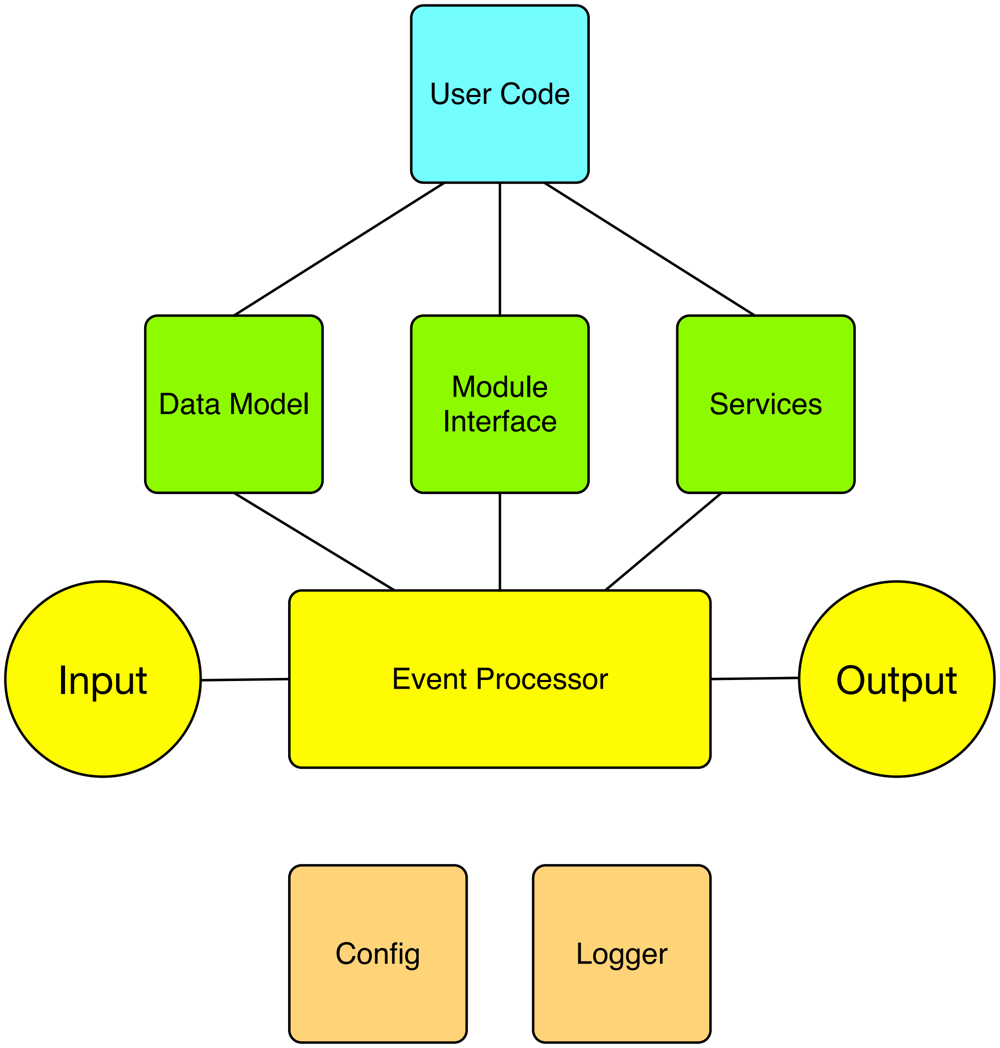
```
// pseudocode! not real C++.
// Part of the body of main
read(infile, &prod1, &prod2);
alg_1(prod1, &prod3);
alg_2(prod2, &prod4);
alg_3(prod3, &prod5);
plots1(prod2, plotfile);
plots2(prod3, prod4,
       plotfile);
write(outfile1,
      prod3, prod5);
write(outfile2,
      prod2, prod4);
```

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.

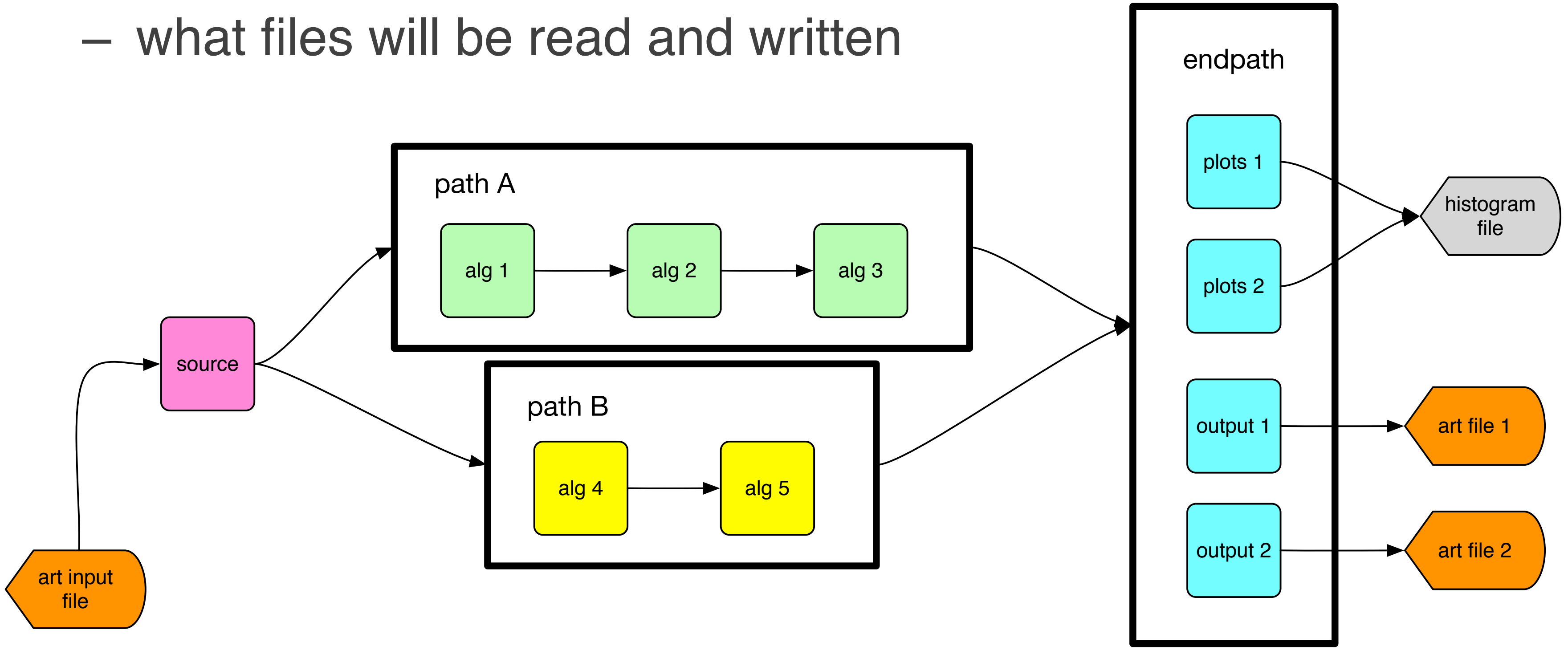
What are the parts of the *art* framework?

- User code is what you and your colleagues 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.



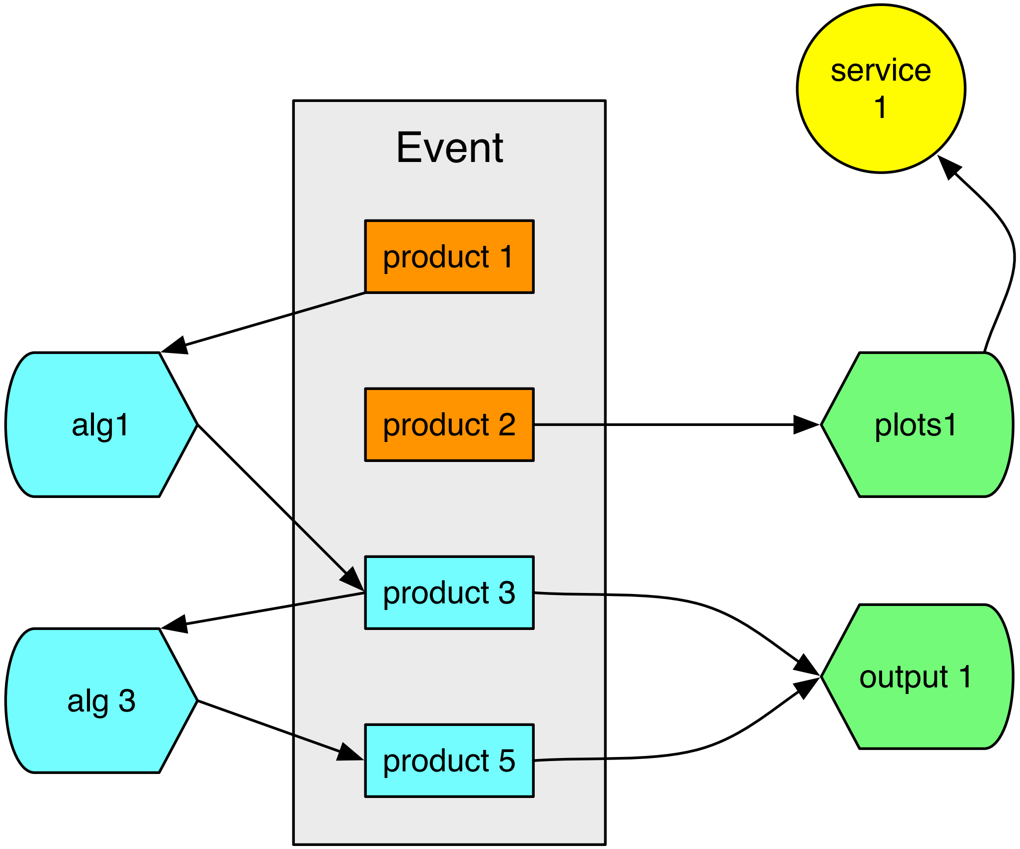
Choosing algorithms to run

- 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).

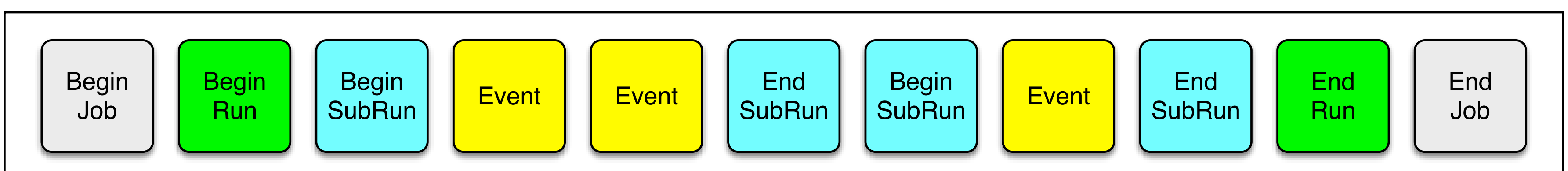


Data: events, subruns, runs, data products

- 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.



Getting input

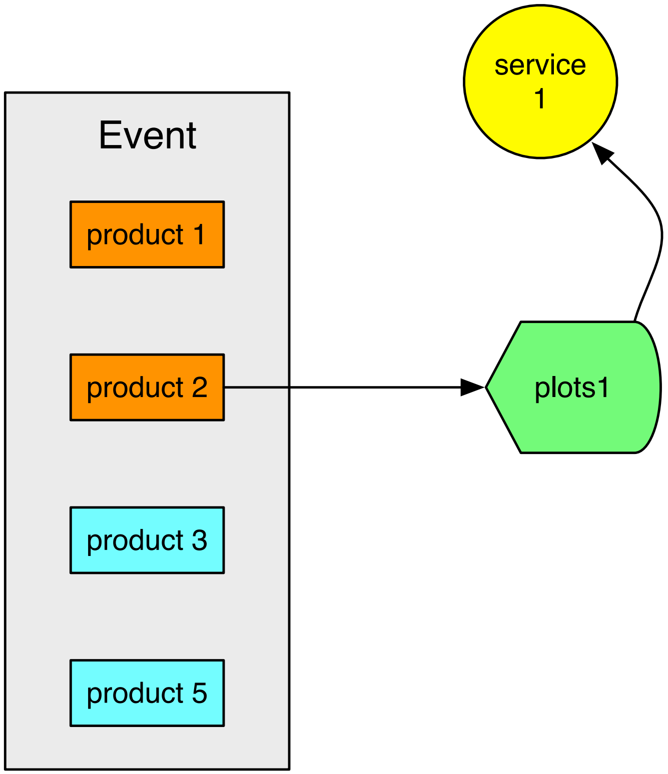
- Sources are the things that tell the framework what *runs*, *subruns*, and *events* are to be processed.
- Some sources read data files (e.g. *RootInput*, which reads the *art*-ROOT data file format, as written by *RootOutput*).
- One source (*EmptyEvent*) creates events containing no products, for use in simulations
- Your experiment may 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

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
- Your experiment will also provide some services
  - Some are provided by LArSoft to many experiments
  - Some are completely experiment-specific
  - examples include access to geometry information, and calibration information

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.



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:

```
std::string greeting { "hello" };
std::string farewell { "goodbye" };
```

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

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 involved

- You're already here. That's a good start.
- Meetings
  - your experiment
  - *art* stakeholders
- Mailing lists
  - art-users@fnal.gov
  - artists@fnal.gov
  - your own experiment will have one or more lists
- Issues (feature requests, bug reports)
  - anyone can report a suspected bug
  - try to get the report into the right tracker
    - experiment code in experiment's bug tracker
    - infrastructure bugs in art issue tracker
  - please discuss feature requests within your experiment, or on the art-users list, before submitting a feature request