Recent relevant LArSoft Efforts

Gianluca Petrillo

LArSoft workshop
Fermilab, June 22–23, 2016

1. Introduction to usability
2. Current effort
3. The future

Outline

1. Introduction to usability

2. Current effort
   - Examples
   - Associations

3. The future

What is LArSoft

LArSoft

A toolkit to facilitate simulation, reconstruction and analysis of events from liquid-argon TPC-based detectors.

Portrait of a LArSoft user:

- runs jobs with existing code...
  $\Rightarrow$ e.g., produces some input special for his task
- ... then uses the result for something new altogether!
  $\Rightarrow$ the LArSoft user is a LArSoft developer

LArSoft content is contributed by: *you*!
You have a determinant role in improving and expanding it.

What is LArSoft

LArSoft

A toolkit to facilitate simulation, reconstruction and analysis of events from liquid-argon TPC-based detectors.

Portrait of a LArSoft user (alternative):

LArSoft content is contributed by: *you*!
You have a determinant role in improving and expanding it.

What is usability, anyway?

usability \ˌyü-zə-ˈbi-lə-tē\ *noun*

"The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use."

— ISO 9241-11

effectiveness: how fully the final solution satisfies the original need
- can you get your idea implemented?
- e.g., signal processing, image processing, MVA...

efficiency: how easy it is to get to that solution
- fitness of the tools
- learning curve
- maintainability

satisfaction: by how many years working with it will shorten your life

Did I say, “maintainability”!

What maintainability has to do with all of this??

... it’s not usability... it’s just code maintainers’ business! *Right?*

Well... no:

- LArSoft is a collaborative contributed project:
  - *you* write it
  - *you* change, fix and extend code to new needs
  - *you* get frustrated when the code is unreadable

  $\Rightarrow$ *your* effectiveness, efficiency and satisfaction are on the table
- maintainability is (also) about
  - *design* accommodating changes $\rightarrow$ *effectiveness*
  - *readable* and understandable code $\rightarrow$ *efficiency*

Outline

1 Introduction to usability

2 Current effort
- Examples
- Associations

3 The future

Area of intervention

- we interviewed numerous users and stakeholders
- collected a list of desirable items...
  - more framework tools
  - easier LArSoft tools
  - more documentation
  - smoother development environment
  - better visualisation and interactive tools
  - ...

⇒ documentation effort (esp. Katherine Lato)

⇒ incremental improvements to building environment (Lynn Garren)

⇒ improvement of LArSoft tools (esp. Saba Sehrish)

⇒ creation of examples (esp. Gianluca Petrillo)

- ... with the active participation of *art* and SCD people
- many of these effort with a limited time (delivery: yesterday!)

Why examples?

- cut and paste shows to be a popular practice in LArSoft
  (hence 41 files in LArSoft `#include <sys/stat.h>`*)*
- when I want to use a tool I am not proficient with, I:
  1. look how others used it
  2. try to adapt their work to my need
  3. end up not understanding why I did what I did
  4. read the documentation to find why it ended up sort-of-working
- a good starting point teaches a lot and fast!

Another place to find examples: the *art* workbook!

- the workbook *should* be an early reading
- LArSoft imposes additional requirements that *art* does not

⇒ our examples illustrate the most demanding of those requirements

How to find what you need

We have:

- a wiki page listing the examples
- a vow to keep it updated

That page also has a guide to find the example you want according to what you are trying to do.

As all things written by "experts", it needs the feedback of real users! Can't find what you look for? the language is unintelligible? Tell us!!

There are a lot of missing items, that will be created *on demand*:

- tell us what you need
- either we'll put together an example,
- or will help you to a solution and will concoct an example out of it

What is in a example

- tried to balance conflicting goals: brevity and completeness
- each example comes with:
  - use of recommended LArSoft practices
  - use of "best" practices (review pending!)
  - unit tests
  - inline documentation in Doxygen format
  - an endless `README` file, hopefully suitable both as user- and reference guide
- this is the shape your final code should also have
  (except you don't need a `README`, but you do need a technical note)

The examples were written with a test-driven development approach:

- write tests first, then "implement" them
- time spent in: writing tests (60%), code (25%) and documentation (10%); debugging (5%)
- felt like I was going nowhere up until two thirds...
- ... and I discovered that at that point I was almost done

Give it a try!

Example: service

We provide *two* examples of services:

- we ask people to follow the factorisation model
- that's a burden, more so for service with multiple implementations
- we provide two examples:
  - a service returning just a number (`AtomicNumber`)
  - a shower calibration service with experiment-specific implementations (`ShowerCalibrationGalore`)

+ learn how to design and write a simple service
+ get explanation side by side with example of the structures for multiple implementations of a service interface
+ see how to write unit tests for services and providers

− no framework events are handled... (good for another example!)

Example: algorithm

A single algorithm example has been added:

- factorisation model is more manageable for an algorithm
  (this model is also endorsed by others, including *art*)
- the only example added:
  - an algorithm returning a list of non-isolated points (`RemoveIsolatedSpacePoints`)

+ learn how to design and structure an algorithm
+ see how to write unit tests for algorithms and modules

± the algorithm itself uses some more advanced C++ techniques

An example for an analyser module was already present.

Testing your code

- good tests are not always easy to design
- writing a test is possibly the single most boring task
- it's also among the most useful ones
- so, I bite it; others will benefit from my pain

Two types:

unit test	C.I. test
small executables	execute `lar`
limited scope, small input	larger scope and input
write C++ code	write a script
add to `CMakeLists.txt`	add to `test_ci.cfg`
run with `mrb test`	run with `test_runner.py`

I wrote a test. Ask me how!

Do you have an idea for a test that you can't turn into code?
Do you want to write a test, but no idea where to start?
Please, with sugar on top: ask us! We want tests *that* much.

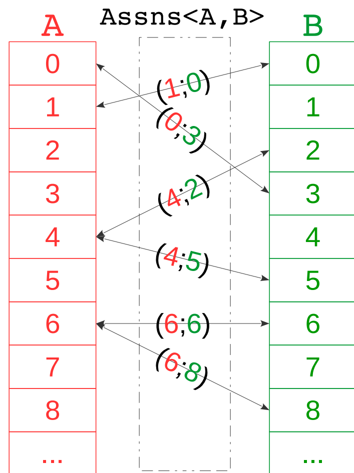
Example: test

An ideal example* of test:

```
 1  // configure a "testing environment" (LArSoft magics)
 2  testing::BasicEnvironmentConfiguration config("ExampleTest");
 3  auto TesterEnv = testing::CreateTesterEnvironment(config);
 4
 5  // set up the service providers the test needs
 6  TesterEnv.SimpleProviderSetup<NeededProvider>();
 7  auto const* neededProv = TesterEnv.Provider<NeededProvider>();
 8
 9  // instantiate the algorithm, set it up
10  Algorithm algo(TesterEnv.TesterParameters("testalg"));
11  algo.Setup(neededProv);
12
13  // run the algorithm, check the result
14  auto result = algo.run(input);
15  if (result != expectedResult) {
16    mf::LogError("testalg") << "everything is wrong!!!\n"
17      << "Got " << result << ", expected " << expected;
18    return 1;
19  }
```

*Smallprint: omitted preparation of input and expected output, assumed FHiCL file configuration and all required service providers support the “simple” setup...

Associations demystified



Associations are:

A set of connections between elements of data products.

You can consider each connection as pair:

```
std::pair<art::Ptr<A>, art::Ptr<B>>
```

... and that is basically it.

Two details of *art* implementation:

- the pair connects *art pointers*
- can't associate objects of same type

Creating associations

Creating an association is as simple as:

```
auto assns = std::make_unique<art::Assns<A, B>>();
//...
assns->addSingle(ptrA, ptrB);
```

Creating associations

It all looks fairly simple... once you have the *art* pointers:

```
1  auto assns = std::make_unique<art::Assns<A, B>>();
2
3  auto handleA = event.getValidHandle<std::vector<A>>(labelA);
4  auto handleB = event.getValidHandle<std::vector<B>>(labelB);
5  //...
6  art::Ptr<A> ptrA(handleA, 1);   // second element in collection of A
7  art::Ptr<B> ptrB(handleB, 0);   // first element in collection of B
8  assns->addSingle(ptrA, ptrB);
```

and we have created the association `(1;0)`.
But...

- ... most of the times *there is no handle*!
- creating the *art* pointer is... unfriendly
- LArSoft's `CreateAssn()` hides it, but asks 5 to 7 arguments...

`CreateAssn()` has been successful, for lack of a better solution.

Creating associations: new approach

We have written a simple utility, `lar::PtrMaker`:

```
1  auto assns = std::make_unique<art::Assns<A, B>>();
2
3  lar::PtrMaker<A> makePtrA(event, module);
4  lar::PtrMaker<B> makePtrB(event, module);
5  //...
6  art::Ptr<A> ptrA = makePtrA(1); // second element in collection of A
7  art::Ptr<B> ptrB = makePtrB(0); // first element in collection of B
8  assns->addSingle(ptrA, ptrB);
```

that takes care of creating *art* pointers to a `std::vector` that is not (yet) saved as data product.

- as simple as with a handle
- still needs to know about event *and producer/filter module*
- might be adopted in *art* if it proves to be useful
- more readable than `CreateAssn()`
- associations, and how to read them afterwards, are still the same!

Outline

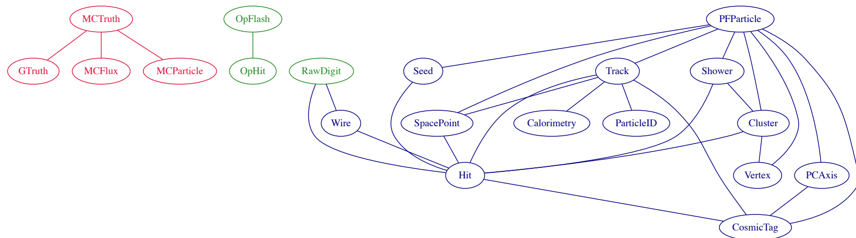
1 Introduction to usability

2 Current effort
- Examples
- Associations

3 The future

Reading associations

- once associations become easy to create...
- ... we'll have herds of them in a event! actually...



- navigation through them may be... challenging
  - e.g., which hits is the track crossing? (track $\rightarrow$ clusters $\rightarrow$ hits)
- we are going to study a heavy user (`AnalysisTree`)
- the goal is to simplify the association query

Developing with canvas

A vision for algorithm development:

- develop in a loosely constrained environment
- migrate algorithm code to LArSoft with no change

The *art* team is delivering such an environment, *canvas*:

- allows reading of input data
- frameworks can be built on top of it (one is *art*)
- or you can use it directly in python or C++

We encourage a “port often to LArSoft” development model!

Don’t miss this afternoon Marc Paterno’s demonstration!!

Ongoing process

From our interviews, a block of ideas came out.
Some that we could not find time and effort to solidify:

- service configuration from input file
- added flexibility to configuration language
- a visual FHiCL navigation tool
- more code examples
- code templates
- full event view
- cleaning of console output
- build system revision (effort ongoing lead by J. Admunson)
- (yet another) event display
- Continuous Integration system interface and features

We are still interested in your feedback.

Comments!

(questions will do, too)

$\rightarrow$ in the breaks during the workshop,

$\rightarrow$ by e-mail,

$\Rightarrow$ *now!*

What is in, what is not

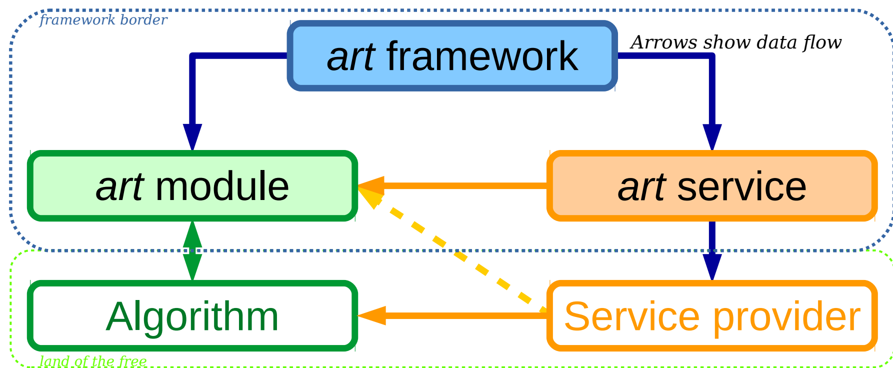
What is where, or when will it be:

- examples: in LArSoft `v05_13_00`
- `lar::PtrMaker`: being tested with a few more modules; trying to wrapping it for delivery

Factorisation model

The idea:

- functional code is independent of any framework
- framework interfaces wrap it to deliver the functionality



See also LArSoft wiki pages about factorised services and algorithms.

Test-driven development I

This is how I did it:

1. plan: write the tests first, then the code to satisfy them
2. the first test reflects how I want to use the provider/algorithm/feature (PAF)
3. include as many corner cases as I can think of (pain here! create input data from scratch, figure out how to check the results, ...)
4. define the PAF class and declare all methods used in the test
5. when declaring each method, document (Doxygen) how it is supposed to be used and to work
6. compile (!); fix every error, until only linker errors from missing methods of my class appear
7. "implement" each method by sentences:

Test-driven development II

```
// TODO determine space partition, cell size, neighborhood

// TODO populate the partition

// TODO for each cell in the partition

// TODO   for each point in the cell

// TODO     compare the point to the ones in the neighborhood
```

8. implement each sentence; if the implementation takes more than three lines, create a new method or class as needed
9. recursion: implement these new constructs by sentences, etc.
10. compile, run the test, debug
11. finish the documentation

CreateAssn

So, what is s wrong with LArSoft's `util::CreateAssn()`?

- it tries to do everything in one line:
  - → there are many versions of it: hard to pick the right one
  - → sometimes the right one does not exist yet: additional juggling needed
  - → many arguments, easy to confuse, forget, swap
- also, it repeats same operations whose results might be cached
- I wanted to provide the same example above for `CreateAssn()`, but I gave up as there is, in fact, no proper version
- if there were, it would probably look like:

```
auto assns = std::make_unique<art::Assns<A, B>>();
//...
util::CreateAssn(module, event, *vectorA, *vectorB, *assns);
```

where `vectorA` and `vectorB` are unique pointers to the collections of `A` and `B`; it would associate the last elements of the two vectors