Three Faces of Wire Cell
Prototype, Toolkit, Integration

Brett Viren

Physics Department

BROOKHAVEN NATIONAL LABORATORY

WC LEE Ana

2017 May 22

Outline

What's in a name

Wire – Cell

some post-hoc philosophizing

wire	embodies a measurement
cell	represents an ideal truth
–	indicates their connection

Three Code Bases of Wire – Cell

Wire – Cell software spans three distinct collections of packages:

1. Wire – Cell Prototype (WCP):
   → initial algorithm development, tryout new ideas, don't worry (too much) about code perfection.
2. Wire – Cell Toolkit (WCT):
   → production quality coding, high performance and long term maintenance.
3. Wire – Cell Integration (WCI):
   → use of WCT components by LArSoft.

- Typically, development of algorithms proceed in order: WCP→WCT→WCI.
- WCP devel may be skipped if WCT-quality design is already understood.
- WCI devel is only needed when new, major **Interfaces** are developed.

Wire – Cell Source Code Organizations

prototype `https://github.com/BNLIF/`

toolkit `https://github.com/WireCell/`

integration `https://cdcvs.fnal.gov/redmine/projects/larwirecell`

- Prototype and Toolkit each consist of a set of multiple `git` repositories.
- Integration consists just the one under the greater `larsoft` UPS umbrella product.

Documentation

prototype `http://bnlif.github.io/wire-cell-docs/`

- Source in Markdown/`mkdocs`, read as HTML.
- Not always maintained, but still useful.

toolkit `https://wirecell.github.io/`

- Source in Org, read as HTML/EPUB/PDF/Markdown/Org.
- Intended to be up to date but not full coverage yet.
- Documentation framework may change.

integration `https://cdcvs.fnal.gov/redmine/projects/larwirecell/wiki`

- Just the wiki, essentially empty.
- We should flesh it out after the next WCI push.

Developer Installation: WCP

Source is aggregated with `git submodules` and built with Waf.

```
git clone --recursive git@github.com:BNLIF/wire-cell.git wcp

cd wcp/ && alias waf=$(pwd)/waf-tools/waf

waf --prefix=/path/to/install configure build install
```

Thereafter, to rebuild/reinstall:

```
waf install
```

See documentation for user install.

Developer Installation: WCT

Source is aggregated with `git submodules` and built with Waf.

```
git clone --recursive git@github.com:WireCell/wire-cell-build.git wct
cd wct/ && alias waf=$(pwd)/wcb
waf --prefix=/path/to/install configure build install
```

Thereafter, to rebuild/reinstall:

```
waf install
```

See documentation for user install.

Pulling, Committing and Pushing Commits

WCP/WCT currently use **git submodules**. So, two step pull:

```
git submodule foreach git pull
git pull
```

To commit and push is also a two step:

```
cd <pkg>/ && hack
git commit -am "My great hack."
git push

cd ..
git commit -am "Pick up update to <pkg>."
git push
```

NB: The **larwirecell** integration code in a single Git repo so apply the usual Git commands. FNAL recommends following git flow

Installation of Software Dependencies - WCP/WCT

WCP and WCP can share same set of “external” packages.
Provide them however it is convenient:

- Install OS packages.
  - Ubuntu has most required packages except ROOT6
- Manually install each from source.
- Automate installation with Spack.
  - This is the recommended approach.
- Use existing binaries
  - Eg, using UPS products at Fermilab
- Some mixture of the above.

Learn how to teach `waf` locations of externals:

```
waf --help
```

Installation: WCI

- WCT is UPS product `wirecell`, WCI is `larwirecell`
- New releases (and their externals) are built to UPS products by FNAL
  - Requests should ultimately go through Redmine tickets.
  - Asking Lynn Garren or LArSoft mgt directly is a good start.
- WCI developer follows "standard" Fermilab way (`ups`, `mrb`, etc).
- **TODO**: we must learn to build `wirecell` UPS product ourselves:
  - Often we have to make fixes to accommodate FNAL problems.
  - Best if we can test full stack at FNAL before making a WCT release.
  - Best if we can make simultaneous WCT+WCI releases.
  - Having FNAL in this test/release loop adds too much delay and confusion.

Ideas to make WCI development easier for us have been proposed but FNAL has responded negatively.

A few random comments about WCP devel

- Coordinate any development with Xin.
  - I'll also try to help, but I give WCP very little attention.
- Do not modify existing code without original author's permission.
  - Use `git blame` or GitHub's `[Blame]` button to find author.
- Okay to copy-and-modify but **must not** reuse same class names.
  - Feel free to make new WCP packages in BNLIF GitHub org.
- Develop prototype as "throw-away" or "first draft" code.
  - Don't get too "attached".
  - WCP and WCT classes and patterns differ but many concepts the same.
  - Long term, code needs to be "ported" to WCT where much **stricter code quality control** will be enforced and only **limited external dependencies** will be allowed in "core" libraries. Preparing for that in WCP makes porting easier.

With that, WCP is a playground, enjoy playing and be creative!

Wire – Cell Toolkit Caveats and Concerns

- There is a lot of "stuff" in Wire – Cell Toolkit.
- Not everything is yet perfectly documented.
- Not everything is even yet perfectly conceptualized or designed.
- **It is ready for contributions from others.**
- However, **please tread lightly**, ask questions, understand intentions.
- Before diving in there should be an informal discussion of scope, design, dependencies, etc.
- Some things can still change, but a lot of things must stand firm.
- I want us to resist making compromises just to satisfy near-term time pressures.

Wire – Cell Toolkit Caveats and Concerns

- There is a lot of “stuff” in Wire – Cell Toolkit.
- Not everything is yet perfectly documented.
- Not everything is even yet perfectly conceptualized or designed.
- **It is ready for contributions from others.**
- However, **please tread lightly**, ask questions, understand intentions.
- Before diving in there should be an informal discussion of scope, design, dependencies, etc.
- Some things can still change, but a lot of things must stand firm.
- I want us to resist making compromises just to satisfy near-term time pressures.

Okay, on to the technical stuff ⟶

Command Line Interface

WCT is a **toolkit** but also provides the wire-cell command line application exposes WCT functionality to the user.

Eg, to run the simulation:

```
wire-cell -c uboone/fourdee.jsonnet
```

Has brief online help

```
wire-cell --help
```

Cmdline arguments that:

- Set one or more "app" **components** to run.
- Load **plugin** libraries that provide components.
- Give one or more **configuration** files.
- Dump hard-coded default configuration.
- Override individual configuation parameters (still todo!)

Package Names and Layout

WCT (and WCP) packages follow fixed layout conventions:

- **name** source package is `wire-cell-<name>`, submodule directory is `<name>/`, shared library, include dir and package name for build system dependencies is `WireCellName`
- `src/` holds `.cxx` implementation and private `.h` headers for a shared library and `wscript_build` connection to build system.
- `inc/WireCellName/` holds public `.h` headers.
- `test/` holds unit test `test_*.cxx` source.
- `apps/` in rare packages, holds code for `main()` programs.

Note: any "public" Python modules/programs should likely be put in `wire-cell-python` package rather than spread among individual packages.

Package Build System

The build system is almost invisible.
Each package needs one file, `wscript_build`, holding a single function declaring the **package name** and connections into **three dependencies trees**:

```
bld.smplpkg("WireCellGen",                           # required
            use="WireCellIface WireCellUtil",        # optional, likely
            test_use="WireCellRootVis",              # optional, likely
            app_use="DYNAMO BOOST")                  # optional, unlikely
```

`use` packages on which the shared library depends

`test_use` (optional) if package's unit tests require more dependencies.

`app_use` (optional) if package has applications and they require additional dependencies.

NB:

- "Core" WCT libraries have very restricted dependencies (eg, no ROOT).
- Most packages **should not** have applications as this is what `wire-cell` is for.

WCT Package Dependency

`wire-cell-*` (implementation packages)
`wire-cell-iface` (interface classes)
`wire-cell-util` (infrastructure and utilities)
external dependencies (Boost, Eigen3, FFTW3)

- Implementation packages **must not** depend on one-another.
- Optional packages may have optional dependencies. Eg, `wire-cell-tbb` implements Wire Cell data flow programming (DFP) using/requiring Intel TBB.
- Some optional dependencies are handled by compile-time switches in the code, Eg. optional Jsonnet support.

WCT Interfaces

Major toolkit functionality is exposed via "**Interfaces**"

- An Interface is an *abstract base class* defining the *virtual methods* to be implemented by a subclass.
- Interface **method arguments** must be:
  - Other Interfaces (typically via `shared_ptr<>`)
  - Classes or types defined in `wire-cell-util`
  - Plain old data types (`int`, etc)
  - STL containers of PoD (`std::vector<float>`, etc)
  - **Must not** reference types specific to another toolkit/framework.

    → no `TH1F`, no `recob::Wire`
- All Interface classes live in `wire-cell-iface`.
- Interfaces have some inheritance hierarchies:

  `INode` data flow programming nodes
  `IData` data products passed between nodes

Data Interfaces

`IData` is base to all data interfaces, provides pointer and vector typed on the subclass (CRTP).

Some examples:

- `IDepo` point energy deposition `(x,y,z,t,q)` with optional transverse and longitudinal extent.
- `IWire` a wire segment in a wire plane leading to a channel.
- `ICell` an association of crossing wires.
- `ITrace` a waveform fragment in a channel at a given time.
- `IFrame` a collection of traces (aka “event”/“trigger”)

There are various more and we can make new ones as needed, but with a lot of thought.

Why Data Interfaces?

It is somewhat unorthodox to have interfaces for data objects.

- As a toolkit, WCT must deal with “foreign” data classes from other applications/frameworks.
  - → Want to avoid unnecessary copies.
  - → Interfaces allow building of facades.
- Do not want to dictate user representations.
  - → One user may have some backing database for some objects.
  - → Another may have load-on-demand files.
  - → A third may want to copy objects bodily.
- Data Flow Programming places some restrictions.
  - → Nodes always pass `shared_ptr` to interface or collections of these shared pointers.

NB: `wire-cell-iface` provides a set of `Simple*` concrete data classes which are trivial bags of data. Implementation code is free to use these if suitable.

DFP Node Interfaces

A Data Flow Programming (DFP) node Interfaces specify:

- data types for node input or output.
- whether node instances are safe to run concurrently.
- the node "category" (DFP connecting behavior, "source", "sink", "function", "queued", etc)

Some DFP node category Interfaces:

- `IFunctionNode` stateless, function-like node.
- `IQueuedNode` object in, collection out, caching.
- `ISourceNode` provide an object each call.
- `ISinkNode` consume an object each call.
- others: `IFaninNode`, `IJoinNode`, `IHydraNode`

NB: Nodes are used even if app does not use full DFP feature.

Typed Node Interfaces

Subclasses of a node category Interface fixes the data types.

- `IDepoSource` source of energy depositions ("depos")
- `IDrifter` queued, drifting of depos.
- `IDuctor` consume depos, produce frames applying induction (field + electronics response).
- `IFrameSource` generate frames (eg, a for a noise source)
- `IFrameFilter` frames in, frames out (eg, digitizer)
- `IFrameSink` consume frames (eg, to writing output file)

Other Interfaces

Some Interfaces simply provide some general functionality.

- `IWirePlane` info about one plane of wires, includes logical wire geometry, field response, vector of `IWire` objects.
- `IAnodeFace` info about wire planes on one face of an anode plane.
- `IAnodePlane` ("APA") info about one or two anode faces, includes wire/channel map.
- `IRecombinationModel` simple function to convert energy deposition to number of drifting electrons.
- `IApplication` a main WCT "app" component with an `execute()` method. (akin to Gaudi `Algorithm` or *art* Module).

WCT Components

A WCT **component** is an **object instance** of an **implementation** (subclass) of one or more WCT **interfaces**.

- Think: *an actor in a play satisfying one or more roles.*
- Some may be small and simple (eg a recombination model) or large and complex (eg, the simulation "app").
- Components may use other components.
- Components are **dynamically constructed**
- They are stored and located via two string labels:

  type: typically (but not necessarily) the implementation class name with any `namespaces` removed. Must be unique.

  name: labels a unique instance of a **type**. If only one instance needed, name is often left blank.

Note: the **type/name** pair is also used in configuration.

`Drifter` - extended example

The "`Drifter`" component from `wire-cell-gen`

- Component **type**: `Drifter`
- C++ class name: `WireCell::Gen::Drifter`
- Definition file: Drifter.h
- Implementation file: Drifter.cxx
- Interfaces: `IDrifter` and `IConfigurable`

Features:

- Is user configurable
- Transports energy depositions in space and time
- Uses a uniform drift velocity vector field.
- Drift stops at a "*field response plane*" near the wires.
- Applies physics: recombination, absorption, diffusion, statistics.

So far it is the only `IDrifter` but there could be others, eg:

- `SCEDrifter` might handle space-charge effects.
- `GArDrifter` might handle gasseous argon drifting.

Drifter - example component definition

Drifter is both an IDrifter and an IConfigurable.

```
#ifndef WIRECELL_DRIFTER   // include
#define WIRECELL_DRIFTER   // guards
#include [...] // snipped for brevity

namespace WireCell { namespace Gen {

class Drifter : public IDrifter, public IConfigurable {
public:
 // IDrifter interface
 virtual bool operator()(const input_pointer& depo,
                         output_queue& outq);

 // IConfigurable interface [snipped for later]

 // ... local methods and data members ...
}; }}
#endif
```

Showing just IDrifter interface method here.

Drifter - example component implementation

```
#include "WireCellGen/Drifter.h"
#include "WireCellUtil/NamedFactory.h"

WIRECELL_FACTORY(Drifter,                                              // (1)
                 WireCell::Gen::Drifter,                               // (2)
                 WireCell::IDrifter, WireCell::IConfigurable); // (3)

//... actual implementation code...
```

A CPP macro is used to “register” the component via args:

1. the **type** label described above.
2. the fully namespace-qualified C++ class name.
3. list of all interfaces this component class implements.

Drifter - example component lookup

```
#include "WireCellUtil/NamedFactory.h"
#include "WireCellIface/IDrifter.h"
using namespace WireCell;

  // ... later in some method ...

  string type = ..., name = ...;

  auto drifter = Factory::lookup<IDrifter>(type, name);
```

- Look up component by **type** and **name**, typically as provided to user code via configuration. Code should **not** “care” their values.
- Component returned as a `shared_ptr<InterfaceType>`, memory management is automatic.
- The component should be considered pre-configured, ready to use.
  - → It is up to the application layer (`wire-cell` or WCI) to assure this.
- Some overhead so do not do `lookup` inside a tight loop!

WCT Configuration Rules

Most WCT components:

- **should be** parameterized to the **greatest extent possible**.
- **must not** use hard-coded numbers!
- **should** provide sane, hard-coded **parameter defaults**.
- **must not** use hard-coded numbers!
- **must not** use hard-coded numbers!

`Drifter` - example configurable

```
#include "WireCellIface/IConfigurable.h"
#include "WireCellIface/IDrifter.h"

namespace WireCell { namespace Gen {

class Drifter : public IDrifter, public IConfigurable {
public:

 // Provide hard-coded "sane" defaults.
 virtual WireCell::Configuration default_configuration() const;

 // Accept config object from user.
 virtual void configure(const WireCell::Configuration& config);
 // ...
}; }}
```

- Inherit from `IConfigurable` in addition to other interfaces.
- Declares `IConfigurable`'s two methods (others not shown).

Code Strategies for Configuration

Driven by:

- You **should** write unit tests for your components.
- Unit tests **may** directly instantiate concrete components.
- Concrete components **may** provide direct configuration methods.
- You **should** avoid separate paths for configuration information.

Recommend following this "configuration flow" pattern:

1. Place "sane" hard-coded defaults as constructor arguments and store as data members.
2. Return default configuration object based on these.
3. Use them as defaults when unpacking a configuration object passed in.

NB: some WCT code may fail to follow this pattern, consider that a bug needing fixing.

Configuration Code Strategy Example

```
namespace WireCell { namespace Gen {

class MyComponent : public IConfigurable {
  int m_a; double m_b;
public:
  // hard-code and store "sane" defaults here
  MyComponent(int a=42, double b=6.9) : m_a(a), m_b(b) {};
  // pass defaults to user
  Configuration default_configuration() const {
    Configuration cfg;
    cfg["a"] = m_a;
    cfg["b"] = m_b;
    return cfg;
  }
  // use as defaults when unpacking user's config object
  void configure(const Configuration& cfg) {
    m_a = get(cfg, "a", m_a);
    m_b = get(cfg, "b", m_b);
  }
  // ...
};}}
```

Note: implementation is in header just for brevity here.

Configuring Components

Manual configuration of component objects is allowed and useful in:

- Writing unit tests.
  - → eg setting parameters directly for brevity.
- Developing integration code.
  - → eg converting from LArSoft's configuration object.

WCT automates component configuring with `ConfigManager`:

- Configuration actions driven by the configuration itself.
  - → Iterates through configuration in strict order.
- Dynamically constructs components as they are discovered in the configuration.
- Matches portion of configuration to each component and applies it.
  - → User config is merged into hard-coded default

Configuration Object Schema

In memory, configuration is a *heterogeneous, recursive* data structure `WireCell::Configuration`.

- It's really just a `Json::Value` (from JSONCPP)
- Top level configuration object is an array (list).
- Each entry is an object (dictionary) with keys **type**, **name** and **data**.
- The **type** and **name** are used to match configuration to configurable component instance.
- The **data** attribute has a value which follows a **type**-specific schema.
- The **data** value is passed to `configure()`.

Example Configuration

Expressed as JSON, with two components both of type `MyComponent` (prior example) and with names `mc1` and `mc2`:

```
[
  {
    "type":"MyComponent",
    "name":"mc1",
    "data": { "a":100, "b":1.0 }
  },
  {
    "type":"MyComponent",
    "name":"mc2",
    "data": { "a":200, "b":2.0 }
  }
]
```

The `name` may be omitted and a default `""` will be assumed.

Two Configuration File Formats

WCT accepts two types of configuration files:

JSON
: as described above, a simple list of configuration objects, one for every configurable component that might be used in an application.

Jsonnet
: a Touring-complete, JSON-like *data templating language* that **compiles into JSON**.

Why two?

- Code wants exhaustive, regular and lightly structured data.
- Humans want concise and highly structured, non-repetitive description of data.

WCT Jsonnet Overview

- Jsonnet is currently **optional** in WCT.
  - If compiled with support Jsonnet files can be directly read.
  - If user's install has no support, can install Jsonnet and use `jsonnet` command line program to compile to JSON.
  - You **really** want to use Jsonnet instead of plain JSON.
- Excellent docs on the Jsonnet web site.
- See also configuration section of the WCT Manual.
- The `wire-cell-cfg` package holds Jsonnet configuration and support.
- Jsonnet is similar in syntax and schema to FCL but it is somewhat more regular and has a super-set of features.

WCT Jsonnet Support

WCT provides `wirecell.jsonnet`

- Exports the WCT system-of-units to Jsonnet.
- Provides Jsonnet forms for common, low-level WCT objects
  - `Point, Ray, Component`
- Helpers for defining DFP graphs.
- Expect it will grow as needed.

Jsonnet Primer

Explore the current config for μBoone sim.

Currently organized in three layers:

- global parameters
- define some default component configurations
- define actual configuration

(click to browse source)

This is just an initial organization. Want to factor out common configuration chunks shared between DUNE and μBoone.

Configuration File Miscellany

- JSON and Jsonnet files are used for other input files:
  - Field response data files.
  - Wire geometry / channel map files.
  - Deposition files
- JSON and Jsonnet files are located first in the current working directory and then by searching a path list given by the `WIRECELL_PATH` environment variable.
- JSON (and not Jsonnet) files may be compressed with `bzip2` (use a `.json.bz2` extension).

Tour of `wire-cell-util`

Bottom of dependencies, it depends on no other WCT package.
Provides **low-level** and **general** classes and functions. Eg:

- `Binning` nbins, maxbin, minbin and related operations.
- `BoundingBox` operations to calculate 3D bounding boxes
- `Point`, `Ray`, `Intersection` 3D vectors
- `ExecMon` = `MemUsage` + `TimeKeeper`
- `String` string functions, eg `split()`.
- `Type` turn C++ types into human readable strings.
- `Pimpos` pitch-impact-position operations
- `Response` field+elec response support.
- `Waveform` for real and complex arrays and FFT.
- Patterns: Interface, Iterator, NamedFactory, Singleton,
- Managers: `PluginManager`, `ConfigManager`

NB: some things probably should become components!

Units

It is **mandatory** that WCT system-of-units is obeyed.
This means any **quantity with units**:

- held in a variable in WCT C++, Python or Jsonnet code **must** be considered expressed in the WCT system-of-units.
- expressed as a literal number in WCT C++, Python or Jsonnet code **must** have a unit **multiplied**.
- **may** be expressed in alternative units if consumed **immediately** (eg, for formatted output) by dividing by a WCT unit symbol.
- expressed as a literal number in JSON or other representation where explicit units are not supported **must** be implicitly in WCT system-of-units.

Units - Literal Values

```
// C++
#include "WireCellUtil/Units.h"
double drift_speed = 1.6*units::mm/units::us;
```

```
# Python
from wirecell import units
drift_speed = 1.6*units.mm/units.us
```

```
// Jsonnet
local wc = import "wirecell.jsonnet";
{ drift_speed: 1.6*wc.mm/wc.us, ... }
```

- Never write a quantity as a bare number without a WCT unit.
- Never have a variable hold a quantity outside WCT system-of-units.

Units - Conversion On Output

May convert units when value is immediately consumed

```
// C++
cout << "Drift speed is " << v/(units::mm/units::us) << " mm/us\n";
```

```
# Python
print ("Drift speed is %f mm/us" % v/(units.mm/units.us))
```

Other contexts to convert to some explicit units:

- Booking histograms to use appropriate ranges.
- Writing output in formats assuming other system of units.
- Note, integration code may hold values in other than WCT system-of-units but must convert when calling WCT code.

Coding Conventions

Covers:

- Formatting (use one the true indent = 4 spaces).
- Commenting conventions.
- Usage of C++ `namespace`.
- Class, method and variable casing and naming patterns.

Exhaustive list in the WCT manual.

Would like to have some editor config to help us comply.

Unit Testing Philosophy

- As you develop some class or functions (units), you write tests.
- Write tests to **fail**, actually test things. Use `assert()` liberally.
- Write tests to be reproducible forever which means no input data nor parameters. For once, it's okay to hard-code!
  - → if input data is truly needed, make it as small as possible so it can be included in the repo, even compiled into the code.
- Tests do not have to be beautiful or high-performance
  - → but they get run a lot so don't make them overblown.
- Tests should be numerous and each should test a small part of some unit.
- Tests should not include large copy-paste from other tests.
  - → Doing this means you are writing "real" code which should be cleaned up and moved into some library (and **it** should be tested)
- Tests are not applications, they should only test, not overloaded into also being "useful".

Writing WCT Unit Tests

1. Start a file under `<pkg>/test/test_name.cxx`.
   - Pick a name that describes the unit (eg, class name)
2. Inside make several `test_<testname>()` functions which take **no arguments** and each which tests some aspect of the unit.
3. Make a `int main()` function that takes **no arguments** and which calls all the test functions.

```
#include "WireCellUtil/Testing.h"
using namespace WireCell;
void test_something() {
  // ...
  int val = some_function();
  Assert(val == 42);
}
int main() {
  test_something();
  return 0;
}
```

Running WCT Unit Tests

- Unit test are built automatically by `waf`.
- `waf` will **try** to run them, but they will fail by default due to environment issues.
- To make it work `waf` needs know how to set a run-time environment. You can set this up in your own shell but that can interfere with build dependencies.

If `$install` points to the installation directory:

```
# first build without running tests
waf --notests build install
# then run all tests using proper harness
waf --alltests --test-cmd="LD_LIBRARY_PATH=$install/lib %s"

# or, run an individual test by hand:
LD_LIBRARY_PATH=$install/lib ./build/<pkg>/test_<name>
```

This needs some improvement!

WCT Python Support

`wire-cell-python`

- Currently a pure-Python package.
- Uses standard Python `setup.py` packaging.
- Wire – Cell system-of-units
- Provides `wirecell` Python module tree and `wirecell-*` main programs using Click CLI.
- Conversion from external config formats (eg, Garfield), generators of config (eg, wires) and tests and plots.
- Various proofs of principle and prototypes.

May extend to allow using WCT components in Python.

Types of Documentation

- Formal manual.
- Doxygen reference.
- GitHub READMEs, issues, pull-requests

Authoring and Building Manual Text

Source at `docs/manual/` and generated content at `https://wirecell.github.io/`.

- Currently written in Org markup language.
- Uses `waf` to build the output formats.
  - Needs Emacs and some Emacs packages installed.
- A few caveats:
  - Emacs is best for writing Org, but not required.
  - The build is currently not tested outside my environment.
  - Some things about the current manual system I don't like and will probably change.

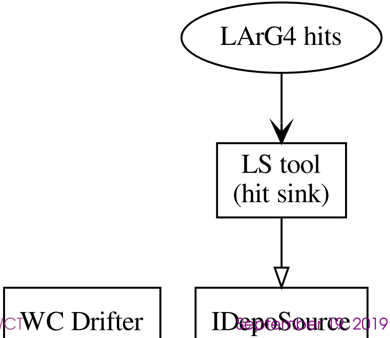
Reference Documentation

- Follow Doxygen commenting rules for any comments that you want to show up in the reference documentation.
- All Interface headers **must** have per-class and per-method Doxygen comments.
  - Some may lack it - this is considered a bug.
- Implementations headers **may** have Doxygen comments.
- Doxygen comments **should not** be used for implementation comments.
- Implementation comments **may** be used.
  - Opinion: Comments always lie. It is far better to spend energy to use **descriptive names** for variables, functions and classes.

NB: Generating Doxygen reference documents is not yet automated and served. Volunteer?

Integration Strategy

- WCT Interfaces and components are similar in concept to *art* Tools
- Any WCT component we expose to LArSoft write an *art* Tool which:
  - Inherits from desired WCT Interface
  - Inherits from LArSoft Tool base class (interface)
- Design *art* Tool side to either accept LArSoft data objects or PoD.
- Some more design details still need to be developed.



Integration Tasks

- Use currently integrated WCT Noise Filtering as vehicle for:
  - Test out the Integration Strategy above.
  - As side effect clean up WCT and WCI parts.
  - Test out ways to handle configuration exchange.
  - This task is for me.
- Integrate existing WCT simulation.
  - This task is for Brian after I provide more concrete guidance based on the above.
- Integrate WCT signal processing once ported from WCP.
  - ditto