# Wire-Cell Toolkit Architecture

## The Basics

Brett Viren

August 31, 2021

# Wire-Cell Basic Topics

- Getting started
- Interface classes
- Components
- Data flow graph
- Applications
- Packaging
- Plugins
- Logging
- Util
- Configuration
- Documentation and community

# Introduction

The Wire-Cell Toolkit (WCT) provides

- A general purpose **component architecture** including an **execution framework**.
- **Implementations** solving various **LAr TPC** problems.
- **Configuration system** with examples for most major LAr TPC detectors.
- **Support data files** (detector description, response function, noise spectra).
- A **command line application**.
- Modular **Python support** package and command line interfaces.
- Extensible **build system** for the toolkit and **user packages**.
- Documentation, and community (GitHub, blog, Mattermost, mailing list)

# Architectural layers of entry to the Wire-Cell Toolkit

- **interfaces** the toolkit API, well-factored abstract base class hierarchy.
- **components** a unit of functionality, implements a number of *interfaces*.
- **aggregation** combine concrete components or via their interfaces to produce arbitrary execution patterns.
- **named factory** dynamically produce an *interface* given the *type and instance* names of its concrete component from *runtime plugins*.
- **data flow** aggregate `INode` *interfaces* into a *data flow graph* to be executed by one of the provided *engines*, also implemented as a *component*.
- **configuration** the `IConfigurable` interface can be fed by the application or by WCT's simple and flexible configuration language.
- **apps** high-level behavior bundled into WCT "app" components.
- **main** top-level (application as a tool) behavior with `Main` class.
- **embed** other applications may call into any of these layers.
- **user interface** toolkit provides the `wire-cell` command line interface.

# Interface classes

An **abstract base class** defines one or more **pure-virtual methods** and describes the **expected behavior** of the implementation.

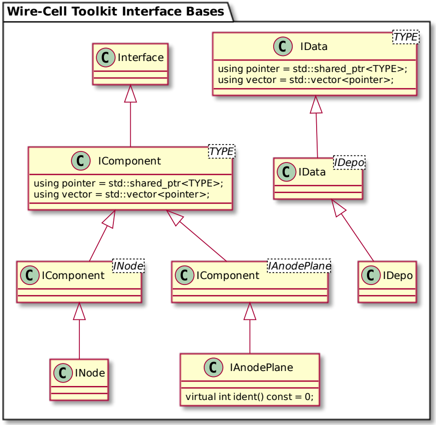
```
class IMethod {
  public:
    // Do something, return result or -1 on error.
    virtual int method(double val) = 0;
};
```

# WCT Interface Library

All “official” WCT interfaces are in a single WCT sub-package:

- **source** `wire-cell-toolkit/iface/`
- **header** `#include "WireCellIface/IMyInterface.h"`
- **library** `libWireCellIface.so`

# WCT Interface Class Hierarchy (roots)




Two branches in the interface class hierarchy:

- *nouns* : all **data interfaces** are from `IData<TYPE>`
- *verbs* : all **component interfaces** are from `IComponent<TYPE>`

The CRTP is used to provide some standard types:

| | |
|---|---|
| `pointer` | a `shared_ptr` to the interface |
| `vector` | a vector of `pointer` |

All interfaces are held by `shared_ptr<>`.

# Some WCT `IData` interfaces

- `IDepo` a localized distribution of ionization electrons.
- `IWire` represent information about one wire segment
- `IChannel` an electronics channel to which wires feed
- `IFrame` dense or spare representation of waveforms.
- `ISlice` a slice in time of a frame.
- `IBLob` a voxel in space with associated value (eg, charge)
- `ICluster` a set of associations between blobs
- `ITensor` a general, dense array of some shape

Some collections of `IData`, themselves `IData` are also defined.

# Components - WCT's verbs

- A **component interface** defines methods that "do" something.
- A **concrete component** (or briefly, just "**component**") is an implementation of one or more component interfaces.

# WCT component categories

**nodes** an INode provides a function-like object that may serve as a vertex in a WCT *data flow graph*.

- Similar to Gaudi Algorithm or *art* Module though the different node types span a variety of interfaces.

**services** an API to some information or process.

- Similar to service or tools from Gaudi or *art* though less constrained.

**features** provide some kind of feature to the component

- *eg*: naming, configuring, finalizing.

A component is usually a **node** or a **service** and not both typically provide one ore more **feature** component interfaces.

# Concrete component inherits from interfaces

```
class MyFilter : public virtual IFrameFilter,
                 public virtual IConfigurable {
  public:
    // IFrameFilter
    virtual bool operator()(const input_pointer& inframe,
                            output_pointer& outframe);
    // IConfigurable
    virtual void configure(const WireCell::Configuration& config);
    virtual WireCell::Configuration default_configuration() const;
}
```

```
// Sketch of the interface class hierarchy

class IFunctionNodeBase : public INode  { ... }

template <typename InputType, typename OutputType>
class IFunctionNode : public IFunctionNodeBase  { ... }

class IFrameFilter : public virtual IFunctionNode<IFrame,IFrame>  { ... }

class IConfigurable : virtual public IComponent<IConfigurable> { ... }
```

# Component factory registration

## In `src/MyFilter.cxx`

```
#include "WireCellUtil/NamedFactory.h"
WIRECELL_FACTORY(MyFilter, WireCell::MyNS::MyFilter,
                 WireCell::IFrameFilter,
                 WireCell::IConfigurable)
```

Factory uses a “type name” (`MyFilter` here) which is technically distinct from but typically chosen to be identical to the component’s C++ class name.

# Method implementation

## The node execution operator

```
bool MyFilter::operator()(const input_pointer& in,
                          output_pointer& out)
{
    out = nullptr;
    if (!in) { return true; }   // more on EOS later
    out = apply_filter(in);
    return true;
}
// configuration related methods described later.
```

Returning `false` is for source nodes to signal they are exhausted.

# Use Named Factory to produce interface instances

## Produce instances of interfaces

```
// One component instance, two interface instances
auto si = Factory::lookup<ISomeInterface>("MyType", "a-name");
auto oi = Factory::find<IOtherInterface>("MyType", "a-name");
si->some_method();
oi->other_method();

// Additional instances of different "type" or "instance" names.
auto si2 = Factory::find<ISomeInterface>("MyType", "another-name");
auto si3 = Factory::find<ISomeInterface>("YourType", "a-name");
```

The production of an instance of an interface is parameterized by:

- C++ interface type,
- The component “type name” and
- An optional component “instance name”.

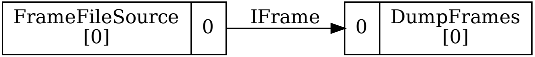
`INode+IConfigurable` components will typically retrieve and hold *service* type components from in their `configure()` method. `IAnodePlane` is a common one to need.

# WCT dataflow graph

WCT can execute INode components aggregated into a dataflow programming (DFP) graph.

- A **node** may consume or produce data through its **ports**.
- A **port** shall pass data of a given type.
- An **edge** transfers data of fixed type from one **output port** to one **input port**.
- A node is in one **category** based on its pattern of ports and behavior.
  - some categories: source, sink, function, fanin, fanout
- Dataflow "programming" means to construct a graph.
  - WCT provides dynamic graph construction driven by configuration.
  - Port type and occupancy rules are asserted during construction.
- A valid DFP graph may then be **executed** by an **engine**.
  - WCT supplies two engines.

# Simple DFP graph




- Graph engine executes source node `FrameFileSource:0`.
- Source produces `IFrame` on its output port 0 and exits.
- Engine transfers `IFrame` on edge and executes the succeeding node.
- `DumpFrames:0` inputs `IFrame` on port 0, does some logging, exits.
- Engine continues until sources are exhausted and edges are drained.

# WCT DFP Stream Protocol

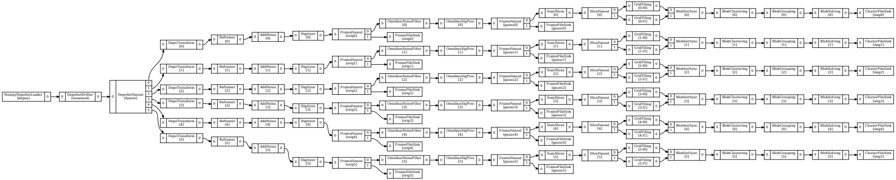
A **stream** is the *sequence of data* seen on an **edge** between two **node ports**. The stream is terminated by a special **end-of-stream** (EOS) marker.

```
stream: ([data(0)], ..., [data(i)], ..., [data(n-1)], [EOS])
```

- Analogous to a C-string of characters terminated with `'\0'`.
  - Each "character" is a `std::shared_ptr<TYPE>`
  - `[data(i)]` holds a non-NULL pointer, `[EOS]` holds `nullptr`.
- A node should flush out any cached data when an EOS is input.
- A node with EOS input shall output a corresponding EOS.
- A new stream may follow an EOS.

When a source is *fully exhausted* of streams, it's next execution after the final EOS shall return `false`. The engine should then not execute the source node again and the graph will begin to drain. Graph execution then terminates.

# Larger example: sim, sigproc, 3D imaging, file I/O



- `IDepoSet` data is read from file, drifted and 6-way *fanned* out.
- Per-APA pipelines implement simulation, signal processing and 3D imaging.
- Intermediate 2-way fanout forwards `IFrame` down pipeline and to `FrameFileSink`, one each for "`orig`" and "`gauss`" frames.
- Each pipeline is capped to save the final `ICluster`.

# WCT has two graph execution engines

`Pgrapher`

- Single threaded, executes only one node at any time.
- Available in core WCT with no extra dependencies.
- Executes graph in reverse topological order, minimizes memory usage.
- Some graphs suffer a speed pathology for $\mathcal{O}(10^5)$ `IData` or more.

`TbbFlow`

- Multi-threaded, execute nodes in parallel.
- Requires WCT built with TBB.
- Parallelism limited by a given max number of threads.
- Efficient even with 1 $\mu$s node execution times.
- Allows multiple data "in flight" at once, higher total memory usage.

Both are identical in terms of their configuration and use.

# WCT applications vs WCT apps

Confusingly, there are two similar terms meaning totally different things:

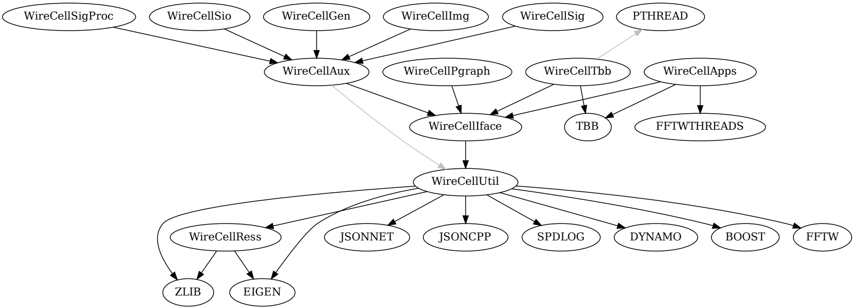
“application” WCT can be embedded in some “external” application.

- WCT provides the `wire-cell` command line program as a simple example of embedding `Main` into a “Wire-Cell Toolkit application”.
- The `larwirecell` package (part of LArSoft) provides an *art* “tool” called `WCLS_tool` that calls `WireCell::Main` and an *art* “module” that can operate the “tool”. The package also provides WCT components that know how to handle LArSoft data.

“app” WCT “internally” defines an “app” (confusingly as an `IApplication` interface) to represent any kind of high-level execution.

- the `Main` class can execute zero or more “apps”.
- `Pgrapher` and `TbbFlow` have been introduced.
- Others include the little used `ConfigDumper` and `NodeDumper`.

# Package dependency hierarchy




Non-WCT external dependencies

`WireCellUtil` low-level utility code used by all WCT.

`WireCellIface` all "official" interfaces.

`WireCellAux` mid-level utility based on interfaces.

`WireCell*` (all the rest) Wire-Cell Toolkit plugin libraries.

Graph is made based on local build config (eg, `WireCellRoot` not shown).

Gray lines are dependency through the unit tests (not libraries).

# Tour of select WCT sub-packages

| | | |
|---|---|---|
| c | WireCellGen | TPC noise and signal simulation. |
| c | WireCellSigProc | TPC noise filtering and signal processing. |
| c | WireCellImg | TPC 3D imaging. |
| c | WireCellPgraph | single-thread data flow graph execution. |
| | WireCellTbb | multi-thread data flow graph execution. |
| c | WireCellSio | "simple" I/O, Numpy, tar, JSON. |
| | WireCellHio | HDF5 based I/O. |
| | WireCellPytorch | DNNROI TPC signal processing. |
| | WireCellZio | experimental ZeroMQ services. |
| | WireCellRoot | ROOT I/O and ROOT-based unit tests. |

- Roughly categorized by "topic" and/or by a major, optional dependency.
- Those marked "c" are considered in the "core WCT", not requiring optional dependencies.

# Wire-Cell Toolkit packages

Name:

- `wire-cell-toolkit/<name>/` for an “official” WCT package
- `any-thing-you-want/` for any WCT “user” package in separate repo

Layout under `/<name>/`:

- `src/*.cxx` holds library source and private header files.
- `inc/` holds public library headers under `WireCell<Name>/*.h`
- `test/test_*.cxx` unit test programs.
  - Also: `test*.sh`, `test*.py`, `test*.jsonnet`.
- `apps/<program>.cxx` source for `program` providing `main()` (rare).
- `wscript_build` brief package build info. Example:

```
$ cat gen/wscript_build
bld.smplpkg('WireCellGen', use='WireCellAux')
```

# Wire-Cell Toolkit “user” packages (WCUP)

A WCUP is simply a WCT-like sub-directory in its own repo.

- Possible to add to `wire-cell-toolkit` proper at some later time.
- No WCT library shall depend on a WCUP library
- A WCUP library shall only depend on WCT via `WireCellAux` and `WireCellIface` libraries.

Examples of WCUPs:

- https://github.com/brettviren/pcbro
- https://github.com/wirecell/wire-cell-gen-kokkos

The `moo` program has a WCUP skeleton generator.

- https://brettviren.github.io/moo/wcup.html

# WCT plugins

A WCT plugin is any shared library containing WCT components.

For automated loading, WCT must be told about all plugin libraries by name either via configuration or the command line:

```
$ wire-cell -p MyPlugin [...]
```

Plugin names: `MyPlugin` is provided by `libMyPlugin.so`.

# Logging in WCT components

```
// MyComponent.h
#include "WireCellAux/Logger.h"
class MyComponent : public WireCell::Aux::Logger, ... {
    // ...
    size_t m_count{0};
    double m_var{0};
}

// MyComponent.cxx
MyComponent::MyComponent()
  : WireCell::Aux::Logger("MyComponent", "pkg") { ... }

void MyComponent::some_method() {
    // ...
    log->debug("call={} var={}", m_count, m_var);
 }
```

We give the **type name** used by named factory and a **logging group** name (usually the short package name: “gen”, “sigproc”, “img”, etc).

# Tracing

Very verbose logging can use CPU even if its log is not emitted.
Embed very noisy log generation in a CPP macro that can be disabled at compile-time.

```
// from img/src/BlobClustering.cxx
SPDLOG_LOGGER_TRACE(log, "got {} blobs, holding graph with {}",
                    blobset->blobs().size(),
                    boost::num_vertices(m_grind.graph()));
```

Of course, do not put side-effects inside this macro!

# Log level guidelines

WCT uses `spdlog` which has “sinks” with ranked “levels”.
In order of desired decreasing verbosity:

- **trace** more than one call per “event” for any given component.
- **debug** O(1) call per “event” from any given component.
- **info** O(1) call from entire job run, communicate some end-result to the user (likely rare to actually use).
- **warn** a rare, non-fatal problem related to some specific input.
- **error** emit just prior to handling some rare but expected error.
- **critical** emit just prior to throwing exception or returning due to an error that was not handled locally.

- If a component works at smaller scale than “event”, be mindful not to over-emit **debug** and prefer **trace**.
- Consider using a dedicated logging **group** for such overly noisy components.

# Control over logging

CLI via the `Main` class controls log sinks and their levels

- By default, all logging is **off**. User must do something to see logs!

Define a sink to standard out and a lowest level of “debug”:

```
$ wire-cell -l stdout -L debug [...]
```

Define a file sink with special level “trace”

```
$ wire-cell \
   -l noisy.log:trace -l error.log:error \
   -L trace [...]
```

# `WireCellUtil` - the base package

Low level utility code that plugins will directly compile against

- Base `Interface` and `IComponent` and `NamedFactory`.
- Arrays, waveforms, FFT, binning, bounding box.
- Graphs, sets, 3D vectors, coordinate transforms, system of units.
- Ray grid, tiling, solving support for 3D img.
- Exceptions, persistency, configuration, base objects.

# WCT System of Units in C++

In WCT **every** numeric literal **must** be given a unit.

```
#include "WireCellUtil/Units.h"
const double drift_speed = 1.6 * (units::mm/units::ms);

log->("Drift speed is {} parsecs per picoseconds",
      drift_speed / (units::parsec/units::picosecond));
```

- Always **multiple** a unit to a literal to bring the value, or a value just read in from some external source, into the system of units.
- Always **divide** by a unit to express in explicit units.

"Never" use values in any other system of units but if you must, mark the variable with the unit:

```
const double tick = 0.5*units::us;
const double tick_ns = tick / units::ns;
log->debug("the tick is {}ns", tick_ns);
```

# WCT configuration subsystem

- Configuration is given in form of a JSON-like (JsonCPP) object[1].
- `ConfigManager` can parse configuration files and feed results to `IConfigurable` instances.
  - Normally users need not worry about this, `Main` handles it.
- WCT directly supports reading files in Jsonnet or JSON format.

[1] There are plans to transition to `nlohmann::json`.

# Component configuration

```
using namespace WireCell;

// Tell toolkit our default configuration.
Configuration MyFilter::default_configuration() const {
    Configuration cfg;
    cfg["threshold"] = m_threshold;
    return cfg;
}
// Recieve actual configuration from toolkit.
void MyFilter::configure(const Configuration& cfg) {
    m_threshold = get(cfg, "threshold", m_threshold);
}
```

Note: this is expected to change soon to provide schema control and type safety and reduce boilerplate code.

# The configuration sequence

WCT is configured with an dependency-ordered array of config objects:

```
config sequence: [ [cfgobj], ..., [cfgobj] ]
```

Each config object has a standard trio of top-level keys:

```
{
    type: "MyFilter",
    name: "a-name",
    data: {
        threshold: 1.0,
        offset: 100*wc.us,  // more on units in config later
    }
}
```

type
: the "type name" registered with named factory.

name
: an optional "instance name" for named factory lookup.

data
: the configuration expected by the component type.

# Constructing configuration with Jsonnet

This can be a talk all by itself

- Read Jsonnet's very fine tutorial, stdlib and reference documentation.

WCT-specific provides Jsonnet support files:

- `wirecell.jsonnet` for units, low-level utility functions.
- `pgraph.jsonnet` help constructing DFP graph configurations.
- `vector.jsonnet` vector arithmetic.
- `pgrapher/experiment/*` detector-specific configuration.

# Feeding configuration

Set `WIRECELL_PATH` to include `wire-cell-toolkit/cfg/` or set include on command line as:

```
$ wire-cell [ ... ] \
            -P /path/to/wire-cell-toolkit/cfg \
            -c my-main-config.jsonnet
```

An application may use `Main` for easy feeding of config to WCT

- In *art* / LArSoft, the `WCLS_tool` provides a FHiCL → WCT config path.

# WCT System of Units in configuration

Same rules apply as with C++: **always** give units to numeric literals.

```
local wc = import "wirecell.jsonnet";
local mycfg = {
    drift_speed: 1.6 * (wc.mm/wc.us);
};
```

# Configuration “bulk” data files

Large, generated config required by some WCT components.

- They are typically generated by dedicated, external programs.
  - Some of which may be found in `wire-cell-python`.
- Generation takes too much time to run “live” in a WCT job.

These files are provided in the `wire-cell-data` package:

- Description of **wire geometry** for popular detectors and
- Their pre-calcualted **field responses**.
- Models of **noise spectra** for the simulation.

Also include directory in `WIRECELL_PATH` or with `wire-cell -P [....]`.

# Documentation and community

Main doc page

- https://wirecell.github.io/

Manual

- https://wirecell.github.io/manual.html

Tutorial

- https://czczc.github.io/wire-cell-tutorial/

News “blog”:

- https://wirecell.github.io/news/

Doxygen reference

- https://wirecell.github.io/doxy/html/

Mattermost (chat)

- https://chat.sdcc.bnl.gov/edg/channels/wire-cell

$\mathcal{FIN}$

*backups*

# Getting source

Released archives:
https://github.com/WireCell/wire-cell-toolkit/releases

Or users may use `git`:

```
$ git clone https://github.com/WireCell/wire-cell-toolkit.git
```

Developers should use:

```
$ git clone git@github.com:WireCell/wire-cell-toolkit.git
```

# Dependencies

Required

- Boost
- TBB
- Eigen3
- FFTW
- Jsonnet
- JsonCPP
- spdlog

Optional

- TBB (recommended)
- HDF5 and H5CPP
- ROOT
- CUDA, Kokkos, Torch
- ZeroMQ and related

Providing dependencies is the job of the user. Not described here, but various Docker/Singularity images, Fermilab/UPS products, Spack recipes, etc are available. Most of the dependencies are provided by a reasonable OS such as Debian.

# Installation

## Build and install

```
$ cd wire-cell-toolkit/
$ ./wcb --help
$ ./wcb configure --prefix=/path/to/install [...]
$ ./wcb install --notests
```

Various `--with-*` options can be given to help `wcb` find dependencies.

Now add `/path/to/install/{bin,lib}` directories to your various PATH vars.

# Test the build

### Command line interfaces

```
$ wire-cell --help
$ wire-cell --version
$ wcsonnet --help
```

### Unit tests

```
$ ./wcb --alltests
```

# Why interfaces?

- High-level composition while hiding low-level detail.
  - "I don't care what your class does as long as it follows the interface."
- Low-level implementation ignoring high-level structure.
  - "I don't care how you use my class, I will focus on satisfying the interface."
- Dynamic and in particular, configuration-driven composition.
  - "We must mix and match the same code in different ways and do not want to write more C++ each time we want something new."
- Plugin architecture support.
  - "Simply name my library to use my components, no need to recompile."

# Data as an interface

- Somewhat unique (aka "controversial") compared to other systems.
- Separate usage from data origin and physical representation.
  - Most algorithms should not care about file formats.
  - Transient/persistent fully decoupled.
  - "Smart" data[2] or simple "bags of values".
- No specific need for an "event store" (*eg* as in Gaudi or *art*).
  - `IData` interfaces may be implemented with an "event store" backend.

[2] WCT exploited this by providing a lazy-loading `IFrame` data type. This "saved the day" by fighting otherwise ruinous memory usage due to ROOT overhead and the ProtoDUNE-SP *art* / LArSoft "raw digit" data model that is input to WCT signal processing.