# Wire-Cell Toolkit Technical Overview

Brett Viren

April 10, 2024

# Topics

- Some historical reflections.
- Understanding the **component-interface-plugin** design.
- A tour of prominent components.
- Understanding the **data flow programming paradigm**.
- External interfaces: command line, *art* / LArSoft, AI/ML, files.

# Select Wire-Cell timeline

- 2015-03-15 First commit to BNLIF/`wire-cell` starting what will be known as the **prototype**.
- 2015-07-06 Wire-Cell **toolkit** starts new GitHub org/repo `WireCell/wire-cell-toolkit`.
- 2015-11-05 `WireCell/wire-cell-tbb` starts **MT data-flow graph engine**.
- 2016-04-29 `WireCell/wire-cell-sigproc` starting with **noise filtering**.
- 2017-04-09 Introduction of **Jsonnet as configuration language**.
- 2017-06-27 Start of `OmnibusSigProc` **signal processing** component.
- 2017-07-05 Modern `larwirecel` **LarSoft integration**.
- 2018-09-20 `WireCell/wire-cell-gen` **2D signal simulation**.
- 2019-01-25 The "ray grid" **3D imaging** method variant added to `img`.
- 2019-08-23 Join submodules into **monorepo** at `WireCell/wire-cell-toolkit`.
- 2019-11-15 `wire-cell-toolkit/pytorch` starts with **DNNROI**.
- 2022-03-10 start of post-tiling **3D imaging** porting from prototype.
- 2024-03-19 initial port of **clustering** from prototype.

# Wire-Cell Prototype and Toolkit

## prototype

- Allow for fast-pace development, MicroBooNE-specific.
- Contains: 3D imaging, clustering and pattern recognition.
- Basis for MicroBooNE Wire-Cell physics analysis team.

## toolkit

- Focus on long-term support and detector-generality.
- Contains: simulation, signal processing, 3D imaging.
- Ongoing effort to “port” prototype algorithms, starting with clustering.

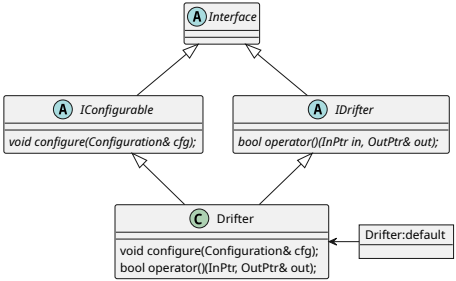
## Shared

- Approximately the same `waf` based build system.
- File based data exchange prototype $\rightarrow$ toolkit.

# Wire-Cell Toolkit foundation: component-interface

**Interface** abstract base class defining method API.

**Component** implementation of one or more interfaces.




```
auto icfg = Factory::lookup<IConfigurable>("Drifter","default");
icfg->configure(cfg);

auto idfr = Factory::lookup<IDrifter>("Drifter","default");
(*idfr)(in, out);
```

# Why Components?

Control complexity making shared commonalities (*ie* the Interfaces).

- Helps to organize and manage large code bases with many contributors.
- Enable new and varied ways of **code composition** without writing new code.

Basic idea was stolen directly from **Gaudi**.

- *art* has similar with its abstract base classes.

WCT goes further with:

- Interfaces define access to "data" (`IData`).
  - (more on `IData` tomorrow)
- Free and open-ended creation of novel `Interface` classes.

# Interface categories and examples

WCT currently has more than 100 Interface classes that fall into these broad categories.

## Data (“noun”)

`IDepo`, `IDepoSet`, `ITrace`, `IFrame`, `ICluster`, `ITensor`

## Nodes (“verb”)

`I*Fan{In,Out}`, `I*{Sink,Source}`, `I*Filter`, `IDrifter`, `IDepoFramer`

## Services / tools / static data (“context”)

`IApplication`, `IDFT`, `IRandom`, `IAnodePlane`, `IWirePlane`

# Prominent toolkit components

## Detector simulation

- `Drifter`: ionization electron **drift** simulation
- `DepoTransform`: ionization electron **signal** simulation
- `AddNoise`: coherent and incoherent **noise** simulation
- `Digitizer`: voltage signal to **ADC** conversion

## Signal reconstruction

- `NoiseFilter`: detector-specific signal pathology and **noise mitigation**
- `OmnibusSigProc`: response deconvolution and signal-ROI **signal processing**

## Event reconstruction

- `GridTiling`: 3D **imaging** (the original "Wire Cell" namesake)
- `MultiAlgBlobClustering`: MicroBooNE **clustering**.
- t.b.d.: ported MicroBooNE **pattern recognition**.

# Configuration constructs and composes components

The toolkit is configured with an ordered list of objects describing component instances.

```
[ // ...
  { type: "FftwDFT" },  // not configurable
  { type: "AnodePlane", name: "apa0", data: {...} },
  { type: "Drifter", name: "default", data: {...} },
  { type: "TbbFlow", data: {...} },
  { type: "wire-cell", data: {...} },
]
```

- An `IConfigurable` instance is given the `data:{}` object as its configuration.
- `IApplication` objects (eg `TbbFlow`) assemble/call other objects.
- A special `wire-cell` type configures top-level `Main` object (and/or use CLI).

Simple list, but must correlate information across its elements in sometimes complex ways.

- Manage complexity by writing in **Jsonnet**, a JSON-like functional programming language.

# Plugin libraries hold components

A component is registered with WCT:

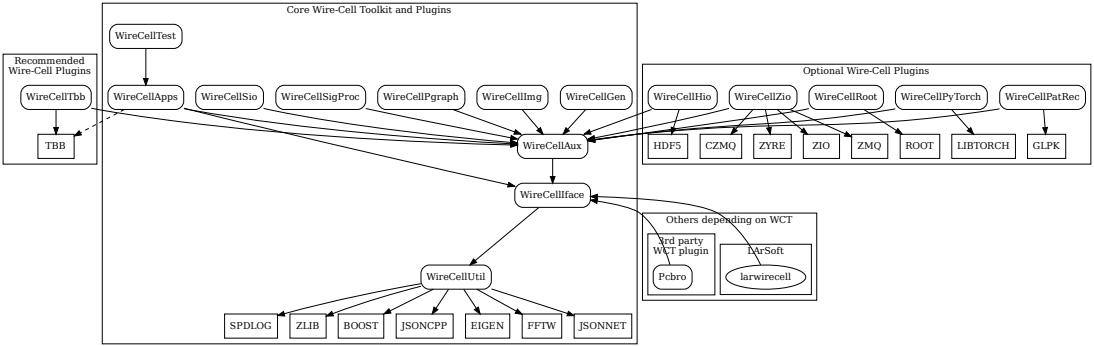
```
// In Drifter.cxx, WCT "type name" is "Drifter", differs from C++ type
WIRECELL_FACTORY(Drifter, WireCell::Gen::Drifter,
   WireCell::INamed, WireCell::IDrifter, WireCell::IConfigurable)
```

Compile component code into a **plugin shared library**, *eg* `libMyPlugin.so`.
And **configure** the job to **dynamically load** its plugins:

```
[   // ...
    {type: "wire-cell", data: { plugins: ["MyPlugin",...] }},
]
```

Developers may independently develop their own plugins and end-users may freely mix/match components from many plugins at run-time.

# WCT dependence - sub-package level view




Tree cleaves into **core**, **recommended**, **optional** and **dependants** (eg LArSoft)

- Core: **util**, **iface** and **aux** and the plugins depending on only them.
- Recommended: but TBB only needed only for **multi-threading** support.
- Optional: less used functionality and requiring more dependencies.

# Ways to use Wire-Cell Toolkit code

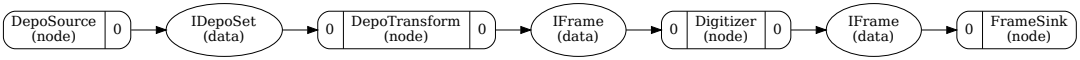
An application may use the toolkit components in many ways:

- Hard-linked "normal libraries" of concrete subclasses/functions.
- As plugins with manually instantiated `Interface`'s via the `Factory`.
- Delegate composition/execution to high level `IApplication` components.
- Embed the `WireCell::Main` high level object (eg as done with the *art* integration).
- Use the `wire-cell` command line program (*ie* the toolkit becomes a framework).

Typical usage: define a **data flow programming graph** in **configuration**.

- With small I/O changes, same graph can run by `wire-cell` or `larwirecell`.

# Data-flow programming




`INode` components represent vertices in a **data-flow graph** exchanging `IData`.

- **Nodes** have numbered input and output **ports** joined by **edges** of specific `IData` type.
- A **graph execution engine** gets graph as configuration, calls nodes and passes data.

Two engines are supplied by WCT:

Pgrapher single-threaded, low-memory.

TbbFlow multi-threaded including possible multiple events “in flight”.

Engines accept a graph description language, can be easily swapped.

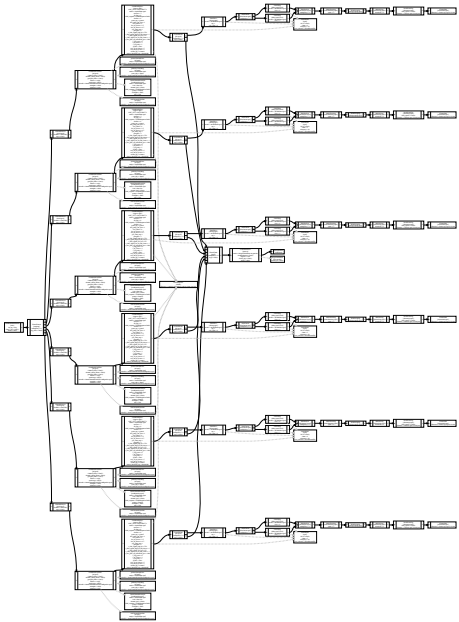
# Example DFP graph - 6-APA ProtoDUNE-SP

Graph generated from actual job configuration $\longrightarrow$

- Noise Filter + Signal Processing + 3D imaging.
  - (details on SP in afternoon session)
- Graph reflects physical PDSP structure.
  - One pipeline per APA.

Possible levels of parallelism with `TbbFlow`:

- Single event at a time: $\approx$ 1 core / APA.
  - Currently, *art* limits to the single-event case,
- Multiple events "in flight": $\approx$ 1 core / node.
  - WCT file input needed for multi-event.



# User command line interface

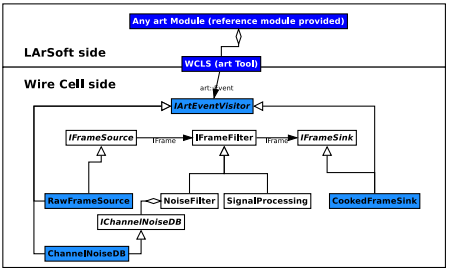
```
$ wire-cell -t 10 -l stdout -L debug -P /path/to/my/cfg \
            -p WireCellGen -p WireCellTbb \
            -A input=depos.npz -A output=frames.npz \
            configuration.jsonnet
```

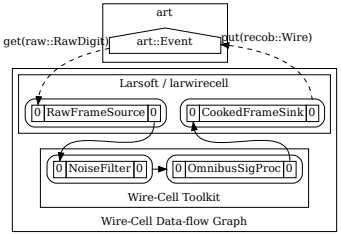
Can control number of **threads** and **logging**, give **file search paths**, load **plugins**, pass configuration **parameters** and name the **configuration** file(s).

- CLI options may also be given in the special `wire-cell` "type" in the configuration file.
- A `WIRECELL_PATH` environment variable may augment file search path.
  - This is essentially only WCT env. var. though `LD_LIBRARY_PATH` consulted to find plugins.

# The *art* + LArSoft interface - `larwirecell`

Class UML and data flow graph.




`larwirecell` provides a **generic interface** to toolkit's `WireCell::Main` object.

- An *art* **tool** wraps `WireCell::Main`, and an *art* **module** to use the tool.
  - Configuration in FHiCL can be converted/injected as WCT Jsonnet/JSON.
- A number of "two-faced" components can "talk" to both the *art* and WCT sides.
  - These provide data converters and wrapped *art* / LArSoft services.

# Sampling of WCT data I/O

## Files

- WCT file formats (`.json`, `.tar{.gz,.bz2}`, `.zip/.npz`)
  - Archives can hold JSON and/or Numpy (`.npy`) files.
- Initial support for direct serialization with HDF5, adding more is a goal.
- Any files supported by *art* / LArSoft via the `larwirecell` interface.

More in the I/O session Thursday morning.

# Wire-Cell Toolkit AI/ML support and strategy

**Files** export to training sets, import from AI/ML generators

- *eg* LS4GAN/UVCGAN detector response translations at ADC level.

**Inference** direct interface to Torch (more in I/O session).

- *eg* to run DNNROI signal processing.

**Development** replace “heuristic” (human learning) function with AI/ML.

- Improve results, more portable across detectors and accelerate with GPU.
- Target blob “deghosting”, clustering and pattern recognition.

More in Friday’s session.

# Links to more info on the Wire-Cell Toolkit

- Main page https://wirecell.bnl.gov/
- GitHub org https://github.com/WireCell/
- Infrequent blog https://wirecell.bnl.gov/news/
- Notes/talks https://www.phy.bnl.gov/~bviren/wire-cell/docs/

*FIN*

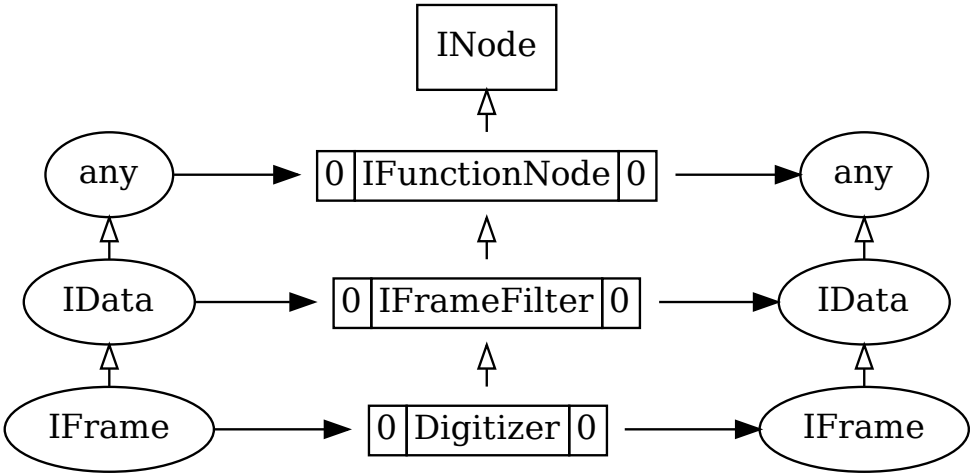
# Wire-Cell Toolkit vs *art* / LArSoft comparisons

There is **not** a competition as WCT runs in *art*. Still, it's natural to want to understand how they may compare.

| Wire-Cell Toolkit | *art* / LarSoft framework |
|---|---|
| Toolkit: provided `main()` is optional. | Framework: provided `main()` is required. |
| Data defined abstract interface class. | Data defined by concrete classes. |
| Distributed data passing on graph edge. | Data marshaled through central `art::Event` store. |
| Minimal data provenance. | Good data provenance tracking and recording. |
| File I/O handled by component/plugin. | File I/O handled by framework. |
| Moderate quality but variety of file formats. | Very good support for ROOT format. |
| MT never runs instance concurrent with itself. | Concurrent MT running instance possible. |
| Arbitrarily fine-grained data flow graph. | Coarse-grain paths (module unit). |
| General graph shapes with explicit branching. | Set graph shapes with built-in branching support. |
| Coarse-grained plugins named explicitly. | Fine-grained plugins searched via name convention. |
| Portable `waf` build. | Semi-portable `cmake` (improvements r.s.n.!). |

Many commonalities: C++-17 (both eager to move to C++-20), share many common external packages (very importantly, TBB), active and dedicated developers, provide critical bits to multiple experiments.

# DFP Data Passing Type Erasure



# Longer Wire-Cell historical timeline

- 2015-03-15 First commit to BNLIF/`wire-cell` starting what will be known as the **prototype**.
- 2015-07-06 Wire-Cell **toolkit** starts new GitHub org/repo `WireCell/wire-cell-toolkit`.
- 2015-10-08 The `wire-cell` command line program starts.
- 2015-10-14 First JsonCPP for configuration objects.
- 2015-11-05 `WireCell/wire-cell-tbb` starts MT data-flow graph engine.
- 2015-12-15 Modern `INode` design started.
- 2016-03-02 `WireCell/wire-cell-cfg` Jsonnet configuration starts.
- 2016-04-29 `WireCell/wire-cell-sigproc` starting with noise filtering.
- 2016-12-20 Initial `larwirecell` module for noise filtering.
- 2017-04-09 Introduction of Jsonnet as configuration langauge.
- 2017-06-27 Actual `OmnibusSigProc` signal processing starts.
- 2017-06-28 `wire-cell` CLI factored out to `WireCell::Main` for
- 2017-07-05 Modern `larwirecel` integration design.
- 2018-03-05 `WireCell/wire-cell-pgraph` starts ST, low mem DFP graph engine
- 2018-09-20 `WireCell/wire-cell-gen` gets today's 2D signal simulation.
- 2018-10-15 The streaming `Drifter` in `gen`.
- 2019-01-07 `WireCell/wire-cell-img` starts with "time slicing".
- 2019-01-25 The "ray grid" 3D "tiling" method invented, added to `img`.
- 2019-02-11 Add ProtoDUNE-SP support to noise filtering.
- 2019-08-23 Join submodules into monorepo at `WireCell/wire-cell-toolkit`.
- 2019-11-15 `wire-cell-toolkit/pytorch` starts with DNNROI.
- 2020-02-28 `wire-cell-toolkit/zio` subpackage starts.
- 2022-03-10 start of post-tiling imaging porting from prototype.
- 2022-04-26 tar/zip files streams and custom Numpy I/O.
- 2022-09-29 start of imaging sampling to point cloud.
- 2024-03-19 port of prototype clustering.