Gaudi::Functional

- Many algorithms are of the type ‘some data in’ → ‘some other data out’
- Standardize this pattern, and factor out ‘getting’ and ‘putting’ the *event* data

- establish a vocabulary
- less code to write,
- more uniform code — easier to understand code somebody else wrote;
- encourage (enforce!) ‘best practice’
- remove ‘boiler plate’ (aka. magic wand waving) from ‘client’ code — so future changes will not affect it! (at least, less likely they will)

Example: RawEvent→FTLiteClusters

templated on ‘signature’: Out(const In&…)

```
struct FTRawBankDecoder
: Gaudi::Functional::Transformer<FTLiteClusters(const LHCb::RawEvent& )>
{

  FTRawBankDecoder( const std::string& name, ISvcLocator* pSvcLocator )
  : Transformer(name , pSvcLocator,
                KeyValue{ “RawEventLocations",
                          concat_alternatives( LHCb::RawEventLocation::Other,
                                               LHCb::RawEventLocation::Default )},
                KeyValue{ "OutputLocation", LHCb::FTLiteClusterLocation::Default } )
  { }
```

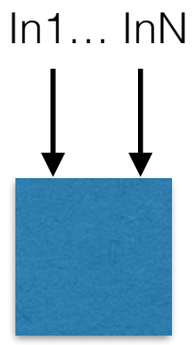
Transformer generates DataObjectHandle<RawEvent>, DataObjectHandle<FTLiteClusters>, and properties “RawEventLocation” and “OutputLocation”

```
  FTLiteClusters operator()(const LHCb::RawEvent& rawEvent) const override
  {
     FTLiteClusters clusters;
     // create clusters from RawEvent
     return clusters;
  }
};
```

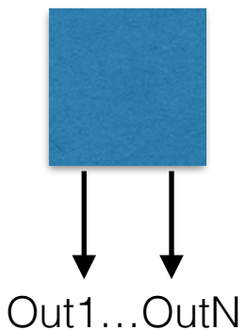
Transformer::execute ‘gets’ RawEvent using data handle, and passes it as ‘const RawEvent&’ to this ‘*const*’ method

and ‘moves’ FTLiteClusters into DataHandle (actually, RVO kicks in, and the move is elided)

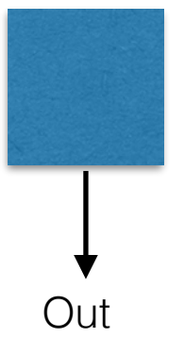
Taxonomy




```
template <typename Signature, typename Traits_ = Traits::useDefaults>
class Consumer;
.
template <typename... In, typename Traits_>
class Consumer<void(const In&…),Traits_>
{
   virtual void operator()(const In&...) const = 0;
};
```



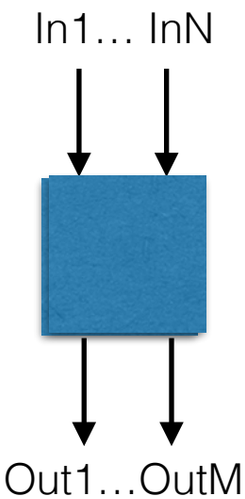

```
template <typename... Out, typename Traits_>
class Producer<std::tuple<Out...>(),Traits_>
{
   virtual std::tuple<Out...> operator()() const = 0;
}
```



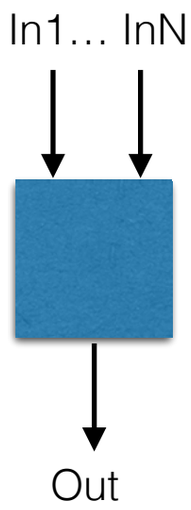

```
template <typename Out, typename Traits_>
struct Producer<Out(),Traits_>
{
    virtual Out operator()() const = 0;
};
```

Each In…, Out… corresponds to a data handle

Taxonomy




```
template <typename ... Out, typename... In, typename Traits_>
class MultiTransformer<std::tuple<Out...>(const In&...),Traits_>
{
   virtual std::tuple<Out...> operator()(const In&...) const = 0;
}
```




```
template <typename Out, typename... In, typename Traits_>
class Transformer<Out(const In&…),Traits_>
{
   virtual Out operator()(const In&...) const = 0;
};
```

Each In…, Out… corresponds to a data handle

Taxonomy

vector<“const In&”> (*)

Out

```
// Many of ‘In’ -> ‘Out’ (#In is known at initialization time)
template <typename Out, typename In, typename Traits_>
class MergingTransformer<Out(const vector_of_const_<In>&),Traits_>
{
   virtual Out operator()(const vector_of_const_<In>& inputs) const = 0;
}
```

In1… InN

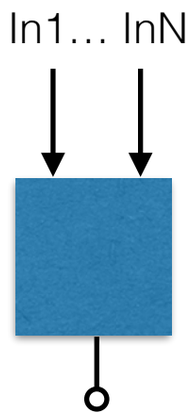
vector<Out>

```
// In1…InN -> Many of ‘Out’ (#Out is known at initialization time)
template <typename Out, typename... In, typename Traits_>
class SplittingTransformer<vector_of_<Out>(const In&…),Traits_>
{
      virtual std::vector<Out> operator()(const In&... ) const = 0
}
```

Each In…, Out… corresponds to a data handle

(*) behaves as either vector<std::cref<In>> or vector<const In*>, depending on std::is_pointer_v<In> — to (dis)allow absent inputs

Taxonomy




```
template <typename... In, typename Traits>
class FilterPredicate<bool(const In&...),Traits>
{
   virtual bool operator()(const In&...) const = 0;
};
```

ScalarTransformer


```
#include “GaudiAlg/Transformer.h"

namespace Gaudi { namespace Functional {

// Scalar->Vector adapted 1->1 algorithm
template <typename ScalarOp,
          typename TransformerSignature,
          typename Traits_ = Traits::useDefaults> class ScalarTransformer;

template <typename ScalarOp, typename Out, typename In, typename Traits_>
class ScalarTransformer<ScalarOp,Out(const In&),Traits_> : public Transformer<Out(const In&),Traits_>
{
    const ScalarOp& scalarOp() const { return static_cast<const ScalarOp&>(*this); }

public:

    using Transformer<Out(const In&)>::Transformer;
    Out operator()(const In& in) const final {
        Out out; out.reserve(in.size());
        auto& scalar = scalarOp();
        for (const auto& i : in) details::insert( out, scalar( details::deref(i)) );
        details::apply( scalar, out );
        return out;
    }
};

}}
```

- Why: less ‘boilerplate’ code, *framework* does the loop for you, and (eventually) ‘chunk’ the loop data, and dispatch ‘chunks’ to tasks. (think eg “tbb::parallel_for”)

ScalarTransformer

‘curious recurring template pattern’

```
struct TTrackFromLong
: Gaudi::Functional::ScalarTransformer<TTrackFromLong, LHCb::Tracks(const LHCb::Tracks&)>
{
  TTrackFromLong(const std::string& name, ISvcLocator* pSvcLocator):
    ScalarTransformer( name, pSvcLocator,
                       KeyValue( "inputLocation", LHCb::TrackLocation::Forward),
                       KeyValue( "outputLocation", LHCb::TrackLocation::Seed)  )
  { }
  using ScalarTransformer::operator();
  boost::optional<LHCb::Track> operator()(const LHCb::Track& trk) const
  {
     auto count = std::count_if( begin(trk.lhcbIDs()), end(trk.lhcbIDs()), isT );
     if (count<5) return boost::none;
     LHCb::Track seed;
     // do stuff with trk to fill seed - not shown here
     return seed;
  };
};
```

```
// general N -> 1 algorithms
template <typename Out, typename... In, typename Traits_>
class Transformer<Out(const In&...),Traits_>
: public details::DataHandleMixin<std::tuple<Out>,std::tuple<In...>,Traits_>
{
public:
    using details::DataHandleMixin<std::tuple<Out>,std::tuple<In...>,Traits_>::DataHandleMixin;

    // derived classes can NOT implement execute
    StatusCode execute() final
    { return invoke(std::index_sequence_for<In...>{}); }

    // instead they MUST implement this operator
    virtual Out operator()(const In&...) const = 0;

private:
    template <std::size_t... I>
    StatusCode invoke(std::index_sequence<I...>) {
        using details::as_const; using details::put;
        try {
            put( std::get<0>(this->m_outputs),  as_const(*this)( as_const(*std::get<I>(this->m_inputs).get())... ) );
        } catch ( GaudiException& e ) {
            (e.code() ? this->warning() : this->error() ) << e.message() << endmsg;
            return e.code();
        }
        return StatusCode::SUCCESS;
    }
};
```

Example

```
template <typename... In, typename Traits_>
class Consumer<void(const In&…),Traits_>
{
   virtual void operator()(const In&...) const = 0;
};
```

Consumer templated on
‘signature’ void(const In&…)

```
class EventTimeMonitor
: public Gaudi::Functional::Consumer<void(const LHCb::ODIN&),
                                     Gaudi::Functional::Traits::BaseClass_t<GaudiHistoAlg>>
{
  EventTimeMonitor( const std::string& name, ISvcLocator* pSvcLocator)
  : Consumer( name , pSvcLocator,
              KeyValue{"Input", LHCb::ODINLocation::Default } )
  {
    …
  }
  void EventTimeMonitor::operator()(const LHCb::ODIN& odin) const override
  {
    …   code using ‘odin’ goes here …
    …   yes, you can fill histograms here! …
  }
```

“Traits” allow customization,
eg. which base class to use.
Default: GaudiAlgorithm

baseclass generates
DataObjectHandle<ODIN>,
and matching property “Input”
with (initial) value as specified

baseclass’ execute ‘gets’ ODIN using data handle,
and passes it as ‘const ODIN&’ to this ‘const’ method

FilterPredicate

```
template <typename... In, typename Traits_>
class FilterPredicate<bool(const In&...),Traits_>
{
       // implement the following operator
       virtual bool operator()(const In&...) const = 0;
};
```

- return value used as ‘filterPassed’ equivalent (i.e. ‘block’ algorithms ‘behind’ it in the control flow graph)
    - ready for the point where ‘filterPassed’ will no longer be ‘state’ of Algorithm
- Examples:
    - Phys/LoKiHlt: {HDR,L0,ODIN}Filter
    - Phys/LoKiGen: MCFilter
    - Hlt/HltDAQ: HltRoutingBitsFilter
    - Rec/LumiAlgs: Filter{FillingScheme,OnLumiSummary}

(Multi)Transformer

- Given N input (containers) (of varying type) produce 1 output (container)

```
template <typename Out, typename... In, typename Traits_>
class Transformer<Out(const In&…),Traits_>
{
      virtual Out operator()(const In&...) const = 0;
};
```

const In1&, …, const InN& —> Out

- Given N inputs (containers) (of varying type) produce (a tuple of) M output (containers)

```
template <typename ... Out, typename... In, typename Traits_>
class MultiTransformer<std::tuple<Out...>(const In&...),Traits_>
{
       virtual std::tuple<Out...> operator()(const In&...) const = 0;
};
```

const In1&, …, const InN& —> Out1, …, OutM

Notes:

1. In1..InN, Out1…OutM can all be *different* types, but the signature must be known *at compile time*
2. Inputs are passed (again) by const&; outputs by value, and are *moved* (not copied) into event store; and the operator() is const

Multi(Transformer)

- HCRawBankDecoder: 2 inputs -> 2 outputs
- (RawEvent, ODIN) —> (HCDigits, HCDigits)

```
class HCRawBankDecoder
: public Gaudi::Functional::MultiTransformer<
             std::tuple<LHCb::HCDigits,LHCb::HCDigits>(const LHCb::RawEvent&, const LHCb::ODIN&),
             Gaudi::Functional::Traits::BaseClass_t<GaudiHistoAlg> >
{
    HCRawBankDecoder(const std::string& name, ISvcLocator* pSvcLocator)
    : MultiTransformer( name, pSvcLocator,
                { KeyValue{ “RawEventLocations",
                            Gaudi::Functional::concat_alternatives( LHCb::RawEventLocation::HC,
                                                                    LHCb::RawEventLocation::Default,
                                                                    LHCb::RawEventLocation::Other ) },
                  KeyValue{ “OdinLocation",     LHCb::ODINLocation::Default } },
                { KeyValue{ "DigitLocation",    LHCb::HCDigitLocation::Default },
                  KeyValue{ "L0DigitLocation",  LHCb::HCDigitLocation::L0 } } )
    {
       …  other properties declared here  …
    }
    std::tuple<LHCb::HCDigits,LHCb::HCDigits>
    operator()(const LHCb::RawEvent& raw, const LHCb::ODIN& odin) const override
    {
        std::tuple<LHCb::HCDigits,LHCb::HCDigits> output;
          … code to “fill” output from ‘raw’ and ‘odin’ goes here…
        return output;
    }
};
```

c’tor requires 2 ‘KeyValue’ arrays of 2 elements each so it can create the 4 properties needed to specify input/output locations

“search path” gets resolved on first access

{Splitting,Merging}Transformer

- Merging Transformer: N containers of the same -> 1 container
- Splitting Transformer: 1 container -> N containers of the same
- N unknown at compile time, set at configuration time (think “vector<Container>”)

- MergingTransformer: Calo/CaloPIDs: InCaloAcceptanceAlg, Tr/TrackUtils: TrackListMerger,
- SplittingTransformer: Hlt/HltDAQ: HltRawBankDecoderBase

- Note: input *containers* can either be ‘mandatory’ — require a vector of references to const container — or ‘optional’ — require a vector of pointers to const containers as ‘signature’

Anti-Pattern: “ping-pong” calls

- Ghost tool interface:

```
/** consider this the beginning of a new event **/
virtual StatusCode beginEvent() = 0;
/** Add the reference information **/
virtual StatusCode execute(LHCb::Track& aTrack) const = 0;
```

  - to use, *first* call ‘beginEvent’ — tool fetches some event data
  - *then* separate calls for each track of interest, and then ‘modifies’ track (i.e. adds extraInfo field)

- Suggested solution:

```
/** Compute ghost information for a list of tracks **/
virtual std::vector<float> execute(const Tracks& listOfTracks) const = 0;
```

  - call with a list (range) of tracks, return list (range) of ghost values, always fetch event data