# Gaudi Design Discussion

Benedikt Hegner

Gaudi Meeting 7.10.2015

# Discussion Points

1. Re-entrant, non re-entrant algorithms, and handles - pros and cons
2. Syntax of declaring dependencies of algorithms and what users see from the TES
3. Lazy creation of objects vs. full definition at configuration time
4. The role and use case for tools
5. The syntax for defining control flow

For full discussion see:

https://groups.cern.ch/group/gaudi-developers/Lists/Archive/Flat.aspx?RootFolder=%2Fgroup%2Fgaudi-developers%2FLists%2FArchive%2FHandle%20use%20cases%20and%20behaviour&FolderCTID=0x012002001151E94208C9DA41A0D034F5A16B5060

# Point 1 - Reentrant vs. non-reentrant Algorithms

To allow for multiple usage of the same algorithm in multiple events at the same time, there are two possibilities:

Deep-cloning -> higher memory usage (currently in GaudiHive)

Re-entrancy -> constraints on allowed code

Q: Should we have a design that encourages re-entrancy?

# Implications of re-entrancy

DataHandles cannot be members of Algorithms

- When filling them in `sysExecute`* the algorithm instance is being tied to a particular event
- **Current Gaudi doesn't allow re-entrant algorithms**

Information about which data to use has to be passed explicitly to `execute`

Several options available

- Passing event context
- Passing event identifier
- Passing all data items explicitly

(*) `sysExecute` is the framework wrapper around the user-provided `execute()`

# Reentrant and non reentrant alongside

```
class MyAlg :: GaudiAlgorithm {
private:
 ReadDataToken<T> m_token;
…
 execute(EventView& view) {
  auto& data = view.get(m_token);
  data.doSomething();
 }
...
}
```

Passing event view

```
class MyAlg :: GaudiAlgorithm {
private:
 RDH<T> m_handle;
…
 execute(EventID& eventID) {
  auto& data = m_handle[eventID];
  data.doSomething();
 }
...
}
```

Passing event identifier

```
class MyAlg :: GaudiAlgorithm {


execute(DataItem& data) {
  data.doSomething();

 }
...
}
```

Passing data items explicitly

```
class MyAlg :: GaudiAlgorithm {
private:
 edm::EDGetToken<T> m_token;
 analyze(edm::Event& event) {
   edm::Handle<T> handle;
   event.getByToken(m_token,
handle)
 }
...
}
```


```
class MyAlg :: GaudiAlgorithm {
private:
 RDH<T> m_handle;
…
 execute() {
   m_handle->doSomething();
  ...
 }
...
}
```

non-reentrant

nota bene:
omitting the syntax for declaring dependencies

# Reentrant and non reentrant alongside

```
class MyAlg :: GaudiAlgorithm {
private:
 ReadDataToken<T> m_token;
…
 execute(EventView& view) {
  auto& data = view.get(m_token);
  data.doSomething();
 }
...
}
```

Passing event view

Requires tools interfaces to get this additional parameter

No constrains on their dependencies

```
class MyAlg :: GaudiAlgorithm {
private:
 RDH<T> m_handle;
…
 execute(EventID& eventID) {
  auto& data = m_handle[eventID];
  data.doSomething();
 }
...
}
```

Passing event identifier

Requires tool interfaces to get this additional parameter

No constrains on their dependencies

```
class MyAlg :: GaudiAlgorithm {



execute(DataItem& data) {
  data.doSomething();

 }
...
}
```

Passing data items explicitly

Requires tools to accept data directly

Only dependencies allowed that the algorithm writer foresees

# Point 2 - syntax for declaring dependencies

- Consensus on:
  - All dependencies should be configurable
  - Data items are identified by a `DataObjID`
  - Configuration and object to access event store are initialized in one go
- Discussion on where and how to set up the dependencies
  - In the constructor
    - via declare or declareProperty
  - In the initialization phase
    - Needing *this* pointer which is only partially inititalized

# Point 3 - Lazy creation of objects

Q: should we have lazy creation of objects in case they are missing from the TES or should the configuration make sure that the necessary producing algorithms are present?

- Problems of lazy creation:
  - Strict ordering requirements
  - The origin of a certain data item is not immediately obvious
  - Problems may be hidden in smartness of framework
- Problems of putting things in the configuration layer
  - Input data specific job options required
  - More manual intervention
- My proposal - have optional algorithms at configuration level

# Point 4 - the role and use case for tools

A while ago we decided to drop the idea of public tools. Whatever has a state beyond an event or a single algorithm is a `Service`.

Q: Should tools just be configurable functors of a given signature or use higher level functionality of the framework in a generic way?

My personal take on that this choice is just at the discretion of every developer/collaboration. As long as the Gaudi implementation allows both.

# Point 5 - Syntax for defining control flow

Proposal is a syntax based on (Python) operators

Boolean operators - &, |, ~

```
someSequence = filterAlg1 & filterAlg2 | ~filterAlg3
```

Composing expressions:

```
someDecision = someSequence & anotherSequence
```

Lazy evaluation vs. full evaluation:

```
someDecision = seq(seq1 | filterAlg2)
```
(lazy evaluation/implicit)

```
someDecision = parallel(seq1 | filterAlg2)
```
(full evaluation)