l3fanalyze Design Note

Gustaaf Brooijmans, Amber Boehnlein, Zhong Yu
Aug 25, 2000

last modified: August 27, 2000

1 Introduction

Package 'l3fanalyze' is an *examiner* class that monitors interested L3 infomation, i.e. data stored in L3Chunk and L3DebugChunk chunks in events.

To make this document more understandable, here are the specific data that l3fanalyze currently deals with:

- L3Chunk::getPhysToolMap(), returns type of std::map<std::string, std::list<L3PhysicsResults*>>
- L3DebugChunk::getDebugInfo(), returns type of std::map<std::string , L3DebugInfo *>

The strings are names of so called *tool_id*s, which are actually just the refsets in trigger list file. Each *tool_id* is mapped to an array of L3PhysicsResults objects in case of L3Chunk, or to one L3DebugInfo object in case of L3DebugChunk.

The job of l3fanalyze is to get the std::map from two chunks, and fill in ntuple with the infomation accessible from the L3PhysicsResults and L3DebugInfo objects. Note that the objects are actually instances of subclasses of the two base classes.

The simplest implementation is to write a huge analyzeEvent() method and do all the ntuple-filling line by line. However, the decision was made that for each child class of L3PhysicsResults (called physics results class), we write one parallel class (called examiner class) that takes care of filling ntuple with objects of that specific physics result subclass. The reason that physics results classes can't do the ntuple-filling themselves is that they are not supposed to have dependency with ntuple stuff.

Same scheme applies to the debug info classes.

Thus, analyzeEvent() only does the business of determine what physics results or debug info are of interest, what their exmainer classes are. The action of filling them into a ntuple is delegated to objects of examiner classes.

During the evolution of l3fanalyze, this design choice causes an explosion of classes and source files, and complicated relations between them. To add a new examiner class whose essential job is just a few lines of ntuple coding, one spends much more effort than that to write the wrapping stuff to make it fit into the structure. The price of OO is justified only if after the original programmer of this package leaves, the successor finds it is easy to understand, maintain, and extend the package.

2 Package Design

The framework class is `L3Analyze`, which implements hook `analyzeEvent()`. Working with its RCP `L3Analyze.rcp`, in its constructor, it determines the location of the ntuple, sets up examiner objects corresponding to interested tool_ids. In `analyzeEvent()`, the two maps stated above in section Introduction are extracted from the event, for each tool_id, correct examiner object are invoked to fill in ntuple with the physics results or debug info associated with that tool_id by the maps.

Examiner classes are all derived from base class `L3Examiner`, there are two branches for physics results and debug info seperately, started with `L3PhysicsResultsExaminer` and `L3DebugInfoExaminer`. Each examiner class comes with its own RCP file, which provides parameters dynamically to regulate its behavior.

The creation of examiner objects is controlled by `L3ExaminerRegistry`, which can create a examiner given its RCP. Of course, those examiner classes must be registered by `L3ExaminerRegistry` through macro `Register_L3Examiner` before that.

2.1 `L3Examiner`

Class `L3Examiner` defines the interface of all examiners.

```
virtual bool init(HepNtuple*, const edm::RCP&);
virtual bool fill(void *) = 0;
```

`init()` is the virtual constructor that does the initialization. It is called by registry, after the default constructor, which subclasses could also provide if something has to be done in constructor. For `init()`, The `HepNtuple*` is the destination ntuple that will be filled. It's the same ntuple throught out the package, and it would be better if it was designed to provide this ntuple in a more centralized way instead of being passed as argument to all examiners through `init()`. The `RCP` is the RCP file of this examiner object, from which the class finds needed parameters.

`fill()` takes a void pointer as argument, to be as general as possible. The real data type should be `std::list<L3xxxPhysicsResults*>*` for physics results examiners and `L3xxxDebugInfo*` for debug info exmainers. Subclasses of

L3Examiner should use a combination of static and dynamic cast to cast the void pointer to desired data type and make sure the cast is successful.

Each examiner should call init() and fill() of its direct father. The root examiner L3Examiner implemented init() such that a protected memeber HepNtuple *ntuple is set to the ntuple passed in; protected members

```
std::string block;          //block name
std::string prefix;         //avoid stupid column name conflict
std::string whole_prefix;   // "blockname::pre_fix"
```

are set up from the RCP parameters "string block" and "string prefix". Note that L3Examiner::init() will always be called, therefore the RCP file of any exminer must contain the two parameter. As a rule, the RCP file of one examiner must contain all parameters that its ancestors required.

It would be nicer that those protected are changed to private and get read access from accessors.

2.2 Examiner Example: L3ElePhysResultsExaminer

Check examiners-guide-line.txt for detailed instruction of how to add an examiner for a physics results class, in this case, L3ElePhysicsResults.

The source files of this examiner also has more detailed comments for being an example.

The basic logic applies to all examiner classes: in init(), one defines columns in ntuple, and in fill(), one casts the data to the right type, extract interested fields, and fill them into the ntuple.

An examiner also needs to define a RCP file, which itself relies on to get its control parameters. The RCP file is also required by registry to set up the correct examiner object, see later.

Other works include putting it in 'COMPONENTS', registering it in 'Reg_L3Examiners.cpp'.

2.3 The Registry

Header 'L3ExaminerRegistry.hpp' defines an important macro 'Register_L3Examiner' which takes the class name of one examiner as argument, to register the class. Rightnow, all registration is done in file 'Reg_L3Examiners.cpp'; whatever file the registration occurs should be linked to framework executable for successfully creating the examiners. If size of framework executable is concerned, it's better to register each examiner in a seperate file, and link only those files that register desired examiners.

Another important method is

```
L3Examiner::createExaminer(HepNtuple*, const edm::RCP&)
```

since the L3Examiner is a Singleton, the common usage is like

```
L3Examiner::instance()->createExaminer(ntuple, rcp);
```

This method is called by `L3Analyze` constructor to set up examiner objects.

How does the registry know which examiner class to instantiate? In each examiner RCP, there's one parameter like

```
string examiner = "L3ElePhysResultsExaminer";
```

which tells the registry what class to find from registered examiners. Therefore, this RCP parameter must be present in all examiner RCPs.

3 L3Analyze

Finally, with examiners that control how to fill the ntuple, and registry that's in charge of register and creation, the `L3Analyze` can do its job in a fairly simple way.

Usually, adding more examiner classes for more physics results or debug infos does not require any modification of `L3Analyze`. Of course, you have to do lots of other works listed in the doc `examiners-guide-line.txt`.

Anyway, the job of `L3Analyze` is very abstract and high level, concentrating on the outermost logic. The logic can be easily changed or improved without touching the examiners.

3.1 the constructor

Currently, two big things are done in the constructor.

First, it determines the location the ntuple. Rightnow, the package is only for offline, the ntuple could be in a local file, or the ntuple could come from the package `analyze`, when that packge is used together. The ntuple setting is controlled and self-documented in `L3Analyze.rcp`.

In future, when `l3fanalyze` can run online, the ntuple destination should also be set up here in constructor.

The proper finishing actions to the ntuple file is carried out in the destructor.

The second thing the constructor does is to setting up examiners. For example, in the case of physics results in L3Chunk, check out following lines in `L3Analyze.rcp`

```
string PhysicsToolIDs = (
        "ELE_LOOSE",
        "XXX_TOOL",
  )
RCP phys_examiner_ELE_LOOSE = <l3fanalyze, L3ElePhysResultsExaminer>
RCP phys_examiner_XXX_TOOL  = <l3fanalyze, XXX_Examiner_RCP>
```

The string array `PhysicsToolIDs` specifies what kind of tool_ids we are interested to monitor. the `RCP phys_examiner_XXX_TOOL` points to an examiner RCP file, so that `L3Analyze` can request the registry to create examiner through this RCP.

Same thing happens to debug info examiners. The `L3Analyze` now established two maps of `tool_id -> examiner`.

Note that, for each tool_id, there must be a unique examiner RCP, even if two tool_ids correspond to the same examiner class. Otherwise, two identical examiner objects will be active during analysis, this will at least cause the conflict that two columns in ntuple use the same tag name. The prefix mentioned before in section `L3Examiner` is to solve this problem, but you have to have two RCPs which defines different prefixes.

That is, say, if you monitor tool_ids ELE_A and ELE_B, and they all use examiner `L3ElePhysResultsExaminer`, you have to write like this way in `L3Analyze.rcp`

```
string PhysicsToolIDs = (
        "ELE_A",
        "ELE_B",
  )
RCP phys_examiner_ELE_A = <l3fanalyze, ELE_A_phy_examiner>
RCP phys_examiner_ELE_B = <l3fanalyze, ELE_B_phy_examiner>
```

The two RCPs `ELE_X_phy_examiner` should at least differ in parameter `prefix`. They can also differ in other parameters to make two examiner objects behave differently.

3.2 `analyzeEvent()`

The logic here is simple. Take debug info as example, the method extracts `L3DebugChunk` from the event, extracts map of `tool_id -> L3DebugInfo`. The it finds the examiner of the tool_id, and feed the examiner with the L3DebugInfo.

What data type is passed to examiner is a common knowledge of both `L3Analyze` and examiner classes. Currently, data passed to debug info examiners is one L3DebugInfo object.

In the case of physics results, the data passed to examiner is a list of physics results. The examiners of physics results should be aware of that and make variable length column arrays.

4 Support

The original programmer, Zhong Yu, could be accessed through `zhong_yu@mail.com` if anyone has any question of this documentation or the package.