`l3fanalyze` Design Note

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

last modified: April 5, 2004

1 Introduction

Package '`l3fanalyze`' contains *examiner* classes that extract Level 3 information from DST inputs to create a ROOT-tuple output. The controlling framework class is called `L3Analyze`, which interacts with the data stored in `L3Chunks` and `L3DebugChunks` through *physics results* and *debug info* classes, respectively.

The simplest implementation would have been to write one huge method within the framework class to do all the ROOT-tuple filling. However, the decision was made that for each physics results class there would be a parallel *examiner* class responsible for filling the ROOT-tuple with its physics results class's objects. The physics results classes do not contain ROOT-tuple filling methods themselves because we wanted to avoid the ROOT-library dependence that would entail. The same reasoning and structure applies to the debug info classes.

During the evolution of `l3fanalyze`, this design choice caused an explosion of classes and source files. Most of the effort involved in adding a new examiner goes into getting the interacting structure correct, not the actual ROOT-tuple filling. We feel the complexity is justified because the design allows ease of maintenance and expansion of the package even with multiple authors.

2 Package Design

The main code in the `l3fanalyze` package is the framework class `L3Analyze`, which implements the hook `analyzeEvent()`. `L3Analyze` initializes and owns the ROOT-tuple, sets up examiner objects corresponding to the *tool_ids* passed in by the user, and in the `analyzeEvent()` hook runs each examiner in turn for every event in the input data file. `L3Analyze` accesses the input data through one of two methods:

- `L3Chunk::getPhysToolMap()`, which returns type `std::map<std::string, std::list<L3PhysicsResults*>>`

- `L3DebugChunk::getDebugInfo()`, which returns type `std::map<std::string , L3DebugInfo *>`

The strings are *tool_id*s, which appear as `refsets` in trigger list file. Each *tool_id* is mapped to a list of L3PhysicsResults objects in case of `L3Chunk`, or to one `L3DebugInfo` object in case of `L3DebugChunk`.

There is a hierarchy to the examiner classes. The lowest level base class is called `L3Examiner`. Two types of intermediate base classes inherit from `L3Examiner`: `L3PhysResultsExaminer` for the physics-results-type data, and `L3DebugInfoExaminer` for the debug-info-type data. The top level examiners inherit from one of these two intermediate classes. Each examiner class also has its own RCP file, which allows behaviour-regulating parameters to be provided dynamically at runtime.

The creation of examiners is controlled by the `L3ExaminerRegistry` class, through a macro called `Register_L3Examiner`. The name of each possible type of examiner must be hardcoded into the file `Reg_L3Examiners.cpp` before it can be successfully recognized by `L3Analyze`.

2.1 L3Examiner

The base class `L3Examiner` defines the interface of all examiners with two virtual methods:

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

The `init()` method is the virtual constructor that does the initialization. It is called by the registry, after the default constructor, and can be over-ridden in child classes if additional specialized initialization is needed. The first arguement of `init()` (the `HepNtuple*`), provides access to the global ROOT-tuple owned by `L3Analyze` [1] . The second arguement provides access to the class's RCP file for the reading-in of run-time configuration parameters. The registry system assures that a derived class always gets passed its own RCP file and not that of a different derived class or a base class.

The `fill()` method takes a void pointer as argument so that derived classes can define their own pointer type. The intermediate base classes define this arguement as `std::list<L3xxxPhysicsResults*>*` for physics results examiners and `L3xxxDebugInfo*` for debug info exmainers. Top-level derived classes of `L3Examiner` should use type castings to verify the validity of, and properly access the contents of, what is being passed to them through `fill()`.

Each examiner should call the `init()` and `fill()` methods of its parent as an early implimentation step. The base class `L3Examiner` `init()` sets protected member `(HepNtuple*) ntuple` equal to the passed HepNtuple pointer, and uses the passed RCP pointer to access values for the following protected members:

[1] Future package upgrades might consider an alternate method of accessing the global ROOT-tuple. For example, make the ROOT-tuple a static, public member of `L3Analyze`

- `std::string block;` (the block, or 'branch,' name)
- `std::string prefix;` (part of an entry, or 'leaf,' name)
- `std::string whole_prefix;` (== *block* + "::" + *prefix*)

Note that `L3Examiner::init()` should always be called, and therefore the RCP file of every examiner must define at least the two parameters `block` and `prefix`. As a general rule, the RCP file of a child examiner class must contain all the parameters that its ancestor(s) required.

2.2 Examiner Example: L3ElePhysResultsExaminer

Please see the file `examiners-guide-line.txt` for detailed instructions on creating an examiner for a physics results class. The class `L3ElePhysicsResults` is used as an example, and so the source files of this examiner also contain detailed comments.

A basic structure applies to all examiner classes: in `init()`, one defines the examiner's columns in the ROOT-tuple, and in `fill()`, one casts the data to the appropriate type, extracts the desired fields, and captures them into the ROOT-tuple.

An examiner also needs to define its own RCP file, which it relies on to get its control parameters.

Other steps include putting the examiner class name in the '`COMPONENTS`' file, and registering it in the `Reg_L3Examiners.cpp` file.

2.3 The Registry

The header file `L3ExaminerRegistry.hpp` defines an important macro called `Register_L3Examiner`. This macro takes the class name of an examiner as its input argument and registers the class within the `L3Analyze` framework executable. Currently all registration is done in the file `Reg_L3Examiners.cpp`, which must be linked to the framework executable in order to operate. If during development the size of framework executable becomes a concern, one could consider registering each examiner in a seperate file, and then linking only those files register necessary examiners.

3 `L3Analyze`

With the individual examiners controlling the filling of the ROOT-tuple, and the registry that is in charge of examiner creation and registration, the executable `L3Analyze` can do its job in a fairly simple way. Adding more examiner classes does not require any modification of the `L3Analyze` code itself. In the same vein, the logic of `L3Analyze` can be easily changed or improved without touching the examiners themselves.

3.1 The Constructor

Currently, two tasks are done in the constructor. First, the parameterization and creation of the ROOT-tuple is handled. Second, each tool instance is associated with an examiner and the examiners are created.

Right now, this package is intended only for offline use and the ROOT-tuple is written to a local file. Such settings are controlled and self-documented in the file `L3Analyze.rcp`. In the future, if `l3fanalyze` is made to run online, the ROOT-tuple destination should probably be changed. Proper closing of the ROOT-tuple file is carried out in the destructor.

To associate the examiners with the proper tool types, `L3Analyze` makes use of the RCP file `ExaminersList.rcp`. The string `PhysicsToolTypes` holds the names of tool types. `L3Analyze` takes each of these type names and prepends the string "phys_examiner_" to create a map key. The RCP values below the `PhysicsToolTypes` string are then used to associate the key with the proper package (usually `l3fanalyze`) and RCP file. `L3Analyze` then knows what RCP files to look in and generate examiners from.

The same process is followed for debug info examiners, with the prepended string this time being "debug_examiner_". After both type lists are processed, `L3Analyze` contains two maps of `tool_type -> examiner`.

Note that for each tool type, there must be a unique examiner RCP. Otherwise, two identical examiner objects will be active, causing conflict within the ntuple because they would both use the same tag name. The prefix mentioned before in Section 2.1 solves this problem as long as the `prefix` values are unique to each RCP file.

3.2 `analyzeEvent()`

The logic for the method `analyzeEvent()` is simple. For each event, the method first tries to extract the requested version (either "online" or "offline") of the `L3Chunk`. It then runs through its list of physics results examiners and feeds the `L3Chunk` to each `fill()` method in turn. The process is then repeated with the `L3DebugChunk` and the list of debug info examiners. The `analyzeEvent()` method sends a warning message to its error logger if a chunk is not found.

The data type passed to a physics result examiner is different than that passed to a debug info examiner. Currently, debug info examiners handle the information in the chunk as one large `L3DebugInfo` object. In the case of physics results, the data passed to examiner is a list of `L3PhysicsResults`. Therefore the physics results examiners must be capable of handling variable length column arrays.

4 Support

The original programmer is Zhong Yu, who may be accessed through email at zhong_yu@mail.com. Other programmers who are familiar with this package include Jon Hays (email : haysjm@fnal.gov) and Angela Bellavance (email : bellavan@fnal.gov). Additional information is available from the Level 3 web pages.