Prototype of a Trouble Shooter for EmCal

L. Aphecetche, GPS of SUBATECH, NANTES, France

April 16, 1999

1 Introduction

Let's briefly remind here what the problem is. Having the EmCal detector running, one must be able to say if it's working correctly or not. To do so, one must have a piece of code, the TroubleShooter (TS), able of checking EmCal objects to see whether they are deviant from a norm or not. When a problem occurs, the TS rises an alarm. The TS must provide a way to access this alarm by clients that have requested to be warned in case of problem.

2 Goals

2.1 Prototyping the TroubleShooter

Before one can take a decision about a software design, one must be able to evaluate this design. The most convincing way of testing a design is probably to turn it into a working program, even if this program has not all the foreseen functionalities. Such a testing program I hereby call a *prototype*.

2.2 This document

The aim of this document is to explain how I designed the TS, how I implemented this design for the prototype, and how this prototype can be tested. The expected audience is the EmCal collaboration, and in particular the EmCal Online group.

3 Design

To design the TroubleShooter I first tried to find the objects relevant to the problem, let's call them the natural objects. Then I had to introduce some other objects, used to give a structure to the whole thing (what I next call the machinery objects). I also tried not to reinvent the wheel, and used, when applicable, some design patterns.

3.1 The natural objects

3.1.1 Checkers and Checkables

As we want to be able to check EmCal objects, I defined two basic objects : the checkable (i.e. an object that can be checked) and the checker (i.e. an object whose aim is to check a checkable). Those objects have definite and distinct responsabilities : the checkable is responsible for updating its internal state (e.g. an EmCal cell must, at each event, update its ADC values), and the checker is responsible for checking some checkable and for rising alarms in case of problem.

A checkable could have several checkers, and a checker could check several checkables. Furthermore, checkers (and possibly checkables) could be added or removed at runtime. Also, checkers can have different levels of criticality.

The idea is that checkers will be coded by different people, at different times, so the TS should provide kind of a template of such a checker, and also some API (Application Program Interface) to write them (see below the alarm distributor).

3.1.2 Alarms

Checkers rise alarms when they find a problem. Those alarms can have different levels of severity (which is a concept different from the criticality of the checkers -see Terry's mail on phenix-emcal-l on 12(14)-Mar-1999-). An alarm must keep track of the checker that rised it and of the checkable that is faulty. An alarm has a message that is an ONCS message (msg_control class) that could be passed to the ONCS system.

3.2 The machinery objects

3.2.1 Tree structure

I defined a TreeItem object, which is an object that has one parent and several children. Checkers, checkables and alarms classes derive from this TreeItem class, so they can be organized in a tree structure (which is kind of natural for the EmCal structure : sector, supermodule, module...)

3.2.2 Subjects and Observers

Here comes the core of the TS! To be able to add/remove checkers and/or checkable at runtime, to be able to manage easily references to alarms, to be able to let the checkers, checkables and alarms objects loosely coupled, I used the Observer design pattern, and created a SubjectAndObserver class, which represents either a Subject object or an Observer object.

Let's take an example to explain the basic idea of the Observer pattern. Let's A be an object (the subject), and B and C two other objects (the observers) that must be warned when the internal state of A changes. The subject object has a Notify method that is responsible for calling the Update method of every observer which is attached to the subject. Interface of the subject comprises methods to attach/detach observers (the reverse is also true with the EMCSubjectAndObserver, i.e. one can attach/detach subjects from observers).

For more information about the Observer pattern, see the book "Design Patterns : Elements of Reusable Object-Oriented Software" by the "gang of 4" : E. Gamma, R. Helm, R. Johnson and J. Vlissides (Addison-Wesley).

3.2.3 Alarm distributor

The aim of the checker is to set/reset alarms. The operation of setting or resetting an alarm is delegated to an alarm distributor object, so different checkers have the same way of setting/resetting alarms. The alarm distributor API gives also access to the logging facility, but the logging itself is delegated to the alarm logger.

3.2.4 Alarm Logger

When an alarm is created, it is also saved into a log file, and into an alarm pool. The log file has the usual meaning of a log file : every and each particular event in the life of the TS is logged, in particular the creation and deletion of alarms. The alarm pool only contains, at a given time, the current alarms (i.e. the 'ON' alarms). This alarm pool is (read-only) accessible from outside the TS, thus other processes can report on alarms.

4 Prototype

4.1 Class hierarchy

```
--EMCAlarmDistributor

--EMCAlarmLogger*
     |
     |---EMCAsciiAlarmLogger

--EMCAlarmRetriever*
     |
     |---EMCAsciiAlarmRetriever

--EMCDataDistributor*

--EMCNamed

--EMCSubjectAndObserver*
     |
     |----EMCTreeItem*
             |
```

```
             |---EMCAlarm* (derives also from TObject)
             |      |
             |      |---EMCBasicAlarm
             |
             |---EMCCheckable*
             |      |
             |      |---EMCElement*
             |      |      |
             |      |      |---EMCDetector
             |      |      |---EMCSector
             |      |      |---EMCSuperModule
             |      |      |---EMCModule
             |      |      |---EMCCell
             |      |
             |      |---EMCElementView*
             |      |      |
             |      |      |---EMCRootElementView* (derives also from TObject)
             |      |             |
             |      |             |---EMCRootDetectorView
             |      |             |---EMCRootSectorView
             |      |             |---EMCRootSuperModuleView
             |      |
             |      |----EMCProcess*
             |
             |---EMCChecker*
                    |
                    |---EMCElementChecker*
                    |      |
                    |      |---EMCCellChecker
                    |
                    |---EMCProcessChecker

--EMCVirtualTroubleShooter*
     |
     |---EMCBasicTroubleShooter
```

The * indicates classes that are meant to be Base Classes. Some are really Abstract in the C++ sense, other are quasi-ABC, that is they do define some default behavior for some methods, so derived classes are not required to redefine all the methods.

4.2 Implementation notes

Here are some various considerations about the current implementation of the prototype. Please feel free to comment on!

4.2.1 Getting Data

Getting data for the cells is made by the the EMCChecker overloaded EMCCell::FetchData() method, which delegates the job to the EMCDataDistributor class for the moment. I use the ONCS Event API (except that I used EMCtestEventiterator, EMCEvent and EMCPackets classes for my tests...). So switching to real ONCS data should not be a problem, once we have a way of making a mapping between what we get from the packet and the different cells!

4.2.2 Detector Structure and Mapping

For the moment, I didn't consider the differences between PbGl and PbSc. In the current implementation, a PbSc EMCModule contains 4 EMCCell, and a PbGl EMCModule contains only 1 EMCCell. Remember this is only a prototype! It's easy to use derivation to get both EMCPbScModule and EMCPbGlModule for instance. Any input on the right way of doing this is welcome!

Anyway, I would need some input about the cells : I'm no so sure what should their data members be ? Are there some lists about that for both PbGl and PbSc ? For my tests I only use 'fake' EMCCell::Set/GetHighPostSample() methods.

I would also have a request on the mapping. Could we decide to have a unique identifier (UID) of the cells ? I would like something like an integer ranging e.g. from 1 to 25632. We could decide to have a 'geometrical' UID : e.g. starting from sector W0 top-left cell (having South in the back) could be one, and then we increase UID through the first row, and so on.... This would have the advantage of being independant on the mounting of the FEE cards for instance, and could be used in the offline analysis also.

4.2.3 From ROOT containers to STL

Following Martin's advice, ROOT is now only present where really wanted All the classes that need some containers (lists mainly) have been switched from ROOT containers to STL containers, apart from the AlarmDistributor which relies on the ROOT TObjArray class (so EMCAlarm is also a TObject). I intend to test the STL solution for the AlarmDistributor also (e.g. using vector).

4.2.4 Checkers

There's only 2 checkers for the moment : one for checking that a particular process is running (on the same LINUX machine), and another one (fake) to check that the HistPostSample() of a cell if greater that 0 (because my EMCtestEventiterator generates -1 values to simulate errors).

Keeping trace of how we checked things : checkers have a method GetVersion() which is indicated in the alarm log, in order to be able to find back the method used to rise the alarms. This suppose that we are keeping all the versions of the code. CVS does this, doesn't it ?

Please note that of course some work has to be done here to define what the checkers should be. Note also that checkers can be attached to cells but also to modules (e.g. to check whether all 4 towers are dead) or at any depth in the hierarchy.

4.2.5 Alarm log, pool and retriever

This part is the youngest in the prototype code. Everything is ASCII for the moment, and no particular attention has been payed about performance.

The EMCAlarmRetriever classe is the skeleton of an API meant to allow processes independant on TS to read(only) the alarm pool.

The Ascii Log File has a line per creation/deletion of alarms :

```
..
EMCBasicAlarm(1.0)|+|some stuff specific to EMCBasicAlarm
EMCBasicAlarm(1.0)|+|...
...
EMCBasicAlarm(1.0)|-|...
```

The + or - indicates creation or deletion. Note also that the class name of the alarm is indicated together with the version. This to allow the AlarmRetriever to generate back alarm objects.

4.2.6 GUI

ROOT is also present in the GUI classes. Those GUI classes should only be considered as a first shot (you will notice that there's no GUI class for the modules and cells for the moment), and are only there to demonstrate how it is easy (I said easy, not fast) to write them and to have them updated automagically thanks to the Observer pattern !

5 Testing the prototype

Want to use it ? All the stuff is in one big dirty directory for the moment. Splitting in several libraries will come soon(er or later...) OK. Log on RCF. Then ssh to rlnx01. Be sure to have your environment correct, e.g. that `ONLINE_MAIN`[1] is defined, that your PATH comprises the PHENIX egcs directory[2], the PHENIX ROOT directory[3], and that the CVSROOT variable points to `/afs/rhic/phenix/PHENIX_CVS`. As an example, here's my PATH :

[1] (`/afs/rhic/phenix/PHENIX_ONCS/mdc2/online_distribution` for example

[2] `/opt/phenix/egcs/bin/`

[3] `/opt/phenix/egcs/root/bin/root`

```
/afs/rhic/phenix/PHENIX_ONCS/mdc2/online_distribution/bin/Linux.i686.egcs:.:
/phenix/u/aphecetc/bin/scripts:/opt/phenix/egcs/bin:/opt/phenix/egcs/root/bin:
/afs/rhic/oodb/v_5.1/egcs/linux86/bin:/opt/phenix/bin:/usr/local/bin:
/usr/bin:/bin:/usr/X11R6/bin:/cern/new/bin:/usr/local/lsf/bin
```

Then :

```
prompt> source $ONLINE_MAIN/setup.com
prompt> mkdir ~/work
prompt> mkdir ~/ts
prompt> cd ~/work
prompt> cvs co online/TroubleShooter/Base
prompt> cd online/TroubleShooter/Base
prompt> ./configure --prefix=/phenix/u/yourusername/ts
prompt> make

case 1 : don't install the TS, just test it
prompt> cd src
prompt> ./test

case 2 : install the TS
prompt> make install
prompt> cd ~/ts
prompt> setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:`pwd`/lib
prompt> cd bin
prompt> ./test
```

In both cases, you should see a ROOT GUI with 6 PbSc sectors (with their 18 green supermodules), with a red moving supermodule (indicating an alarm in one of the cells). You can double-click on a supermodule to have some more information. The alarms are logged in `ts.log`, and the alarm pool is named `ts.pool`

You can also use the `test_pool` program to test the access to the alarm pool by an extra process :

```
(while being in ~/work/src or in ~/ts/bin)
prompt> test & ; test_pool
```

It can be slow as the `test_pool` process (running on the same machine as test) poll the pool at a high rate... Have a look at the code of `test_pool` if you like, but you should see, before the end of test, a list of 20 alarms on screen, list which was rebuild by `test_pool` from the running pool filled by `test`. To quit type CTRL+C.

6 Going further...

If you want to play around with the test.C program, I would encourage you to try to create a new cell checker, starting from the example code of the EMC-CellChecker class. Please report any success !