# An abstract view of the development of an analysis system for The MAGIC Telescope

J C González
Max Planck Institute for Physics, Munich
<gonzalez@mppmu.mpg.de>

February 20, 2000

**Abstract**

In this document I will to express my personal view on the design of an off-line, or better *non real-time conditioned*, analysis system for the MAGIC Telescope. Several institutes are taking part in this experiment, and a big effort has to be done in coordinating all the people who contribute to this project.

**Contents**



DRAFT

# 1 Introduction

The MAGIC Telescope Collaboration is composed by several institutions in Germany, Spain, Italy, Crimea and Armenia. There are also several persons involved from other countries. Of course, not all the people in these institutes and universities will be involved in the design of the software tools needed for the data acquisition, control and analysis. Nevertheless, a considerable number of people will take part, more or less directly, in the development of these programs. In order to make a success out of it, a clear image of what is needed and what we want is crucial.

Some of the things you will read in this small document may sound trivial. Well, I would say probably most of them *are* trivial. But in designing such a complex system it's much better to go step by step, being sure that one always knows what he/she is doing.

In this small document I will enumerate the list of tasks which we have before us. Moreover, I will try to outline the interrelation between these tasks, their priorities and the way they communicate with each other. This will show just which the strategy I suggest is. The actual design can be and will be for sure different, since there are still a lot of unclear things to be discussed. All this work, to be done, will help us not only to clarify a lot some of those points which seem to be obvious *a priori*, even when they are not, but also will serve as additional documentation in case our system has to be changed or upgraded.

# 2 Steps in the design of the software system

There are several steps in the design of a complex software system. I will assume henceforth, as a clear decision, that an Object Orientation (OO) approach is going to be use in the final system. I will as well assume several things about how the different kinds of data are handled and organized, but this would be at the end a matter of details. As an example, I will talk about control, calibration, and real data files, although in principle we could merge all these data in a single file, or split in several sub-types of files.

In my opinion, the evolution of the design of a complex software system should follow these guidelines:

**Requirements analysis phase** In this phase we shall study what the requirements of the system are and which their different uses. Once we know what our system must do, we should define all the events[1] that eventually will occur in the evolution of the whole system (computing system + agents). The definition of events will help us to identify probable collision of actions, in which case a set of given priorities must be established. Also in this phase we shall identify the different uses (*use cases*) of our system.

**Domain analysis phase** In this phase we must identify the different objects taking part of our system and/or interacting with it. These objects may have different natures, they can be active or passive objects, physical devices or their wrappers, visual elements or even abstract concepts. Once we have defined these objects, we should try to visualize their interdependencies in a global *class diagram*. This class diagram will help us to subdivide our system in autonomous subsystems, and for each of these we shall create a subsystem class diagram.

[1] By using the word *event* I do **not** mean event in the physical meaning we are used to. What I mean by event is simply something that happens in our analysis software, an action that could have a reaction from the system when it is detected (for example a *mouse-click* is a typical example of an event: if the mouse happens to be *clicked* on a button in our interface with a label "Quit", the probable reaction from the system will be to finish itself and end the analysis session).

**Define objects behaviour** Once we have identified the objects inside our system, we must define the dynamical behavior of each of them, in response to the different events which may occur in the system.

**Detailed design** After having defined all the structure of our system globally, we must go into details. This is specially true for any wrapper of a hardware component.

**Implementation** This is the mechanistic step in the design of our system. We will use any object oriented language for our purposes.

**Testing** Although in this list this is the last point, it doesn't mean that this task should be performed at the end. On the contrary, we should use the *clean-room* philosophy, which briefly emphasizes correctness verification, rather than testing, as the primary mechanism for finding and removing errors. I will talk about this issue in more detail in another section.

# 3 Analysis system for MAGIC

I will try to use this methodology in the different steps of the design of a data analysis system for MAGIC. In MAGIC we will have a lot of peripherals providing calibration data, which will be stored in disk and available in the analysis phase. These data will be available, of course, in the near-real-time analysis as well. Since there would be no big difference in the near-real-time analysis and in the off-line analysis, we could aim our design in order to take into account both regimes. However, I will concentrate in the off-line analysis system (or better, *non real-time conditioned* analysis).

## 3.1 Requirements analysis phase

As I mentioned before, in this phase we must study what the requirements ans uses of the system are. In a few words, the main features of our system must be:

- It must load real data files, calibration files, system control files and analysis criteria files, and analyze them.
- It must be able to interpret command files provided by the user.
- It should have online help capabilities (tool-tips, what's this tips, online help files, whatever).
- It must generate a whole set of statistics and results, and save them in a file.
- Capability of printing statistics from the events, or events data.
- Command-line mode, for batch operation (BATCH mode).
- Easy to use with a friendly Graphical User Interface (GUI). The system, when operating in GUI mode, must react to any event from the system itself or from the outside world.
- It must be able to select individual events, in order to extract them or analyze them alone, following several criteria provided by the user.
- Capability of exhaustive logging of every action/reaction of the system.

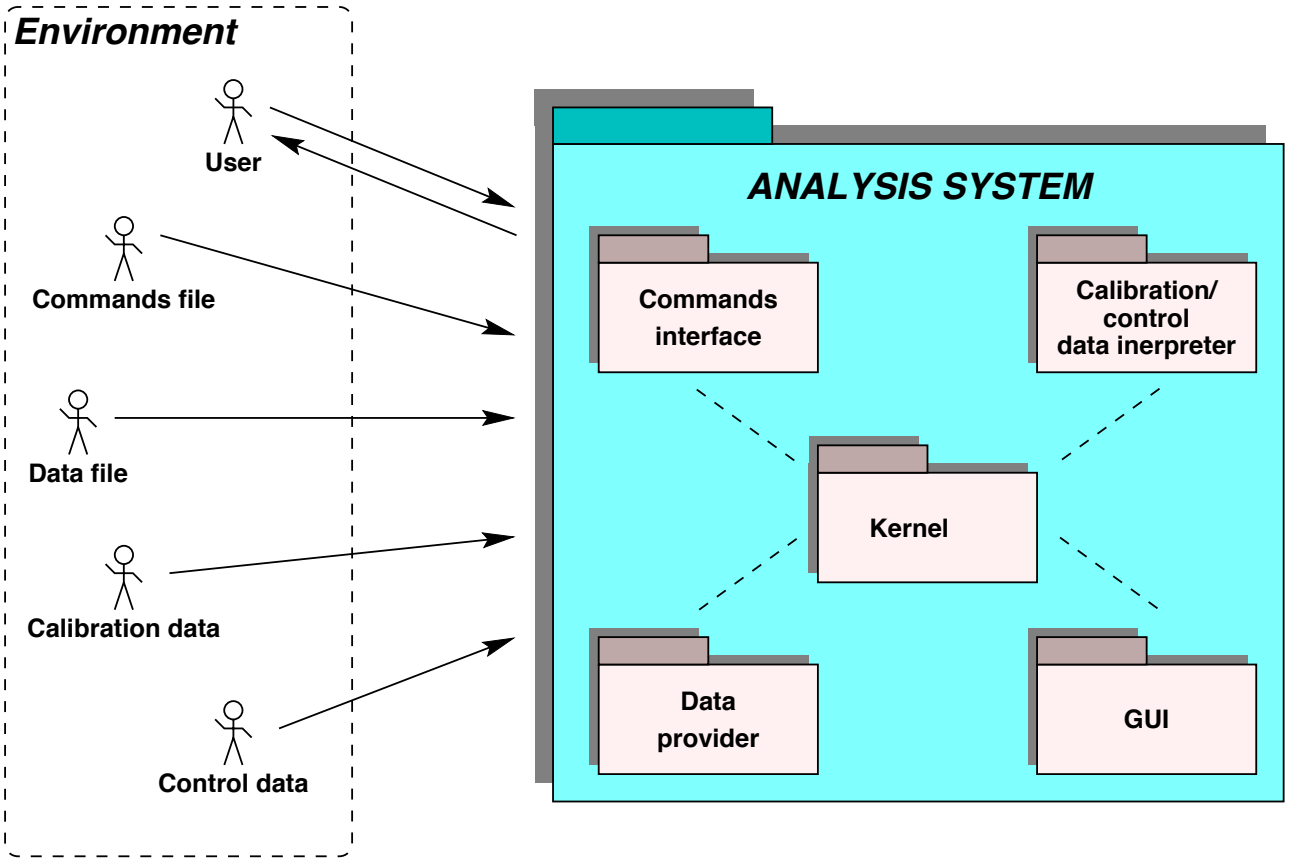


Figure 1: Simple illustration of the communication of our analysis software system with its environment.

This is just a generic list of features. The optimal list of features/requirements of the system can be obtained after discussion, in what is called *kick-off meetings*. Nevertheless, in order to better understand the requirements of our system, we must analyze which events can occur in the context of the operation of our system.

### *3.1.1 Events*

First, we have to define the *environment*. Our environment is defined by several agents. We would have the commands file, if any (when working in BATCH mode), the data files, the calibration files, and the user (when working in interactive, or GUI mode). The system interacts with the environment through messages. These messages can be requests by the agents (the user or the commands file) or services by the system to the agents. In the fig. 1 a general view of such system is shown. From these messages, the most important ones are those which I called *events*. I will try now to identify some of the events our system should support.

i. The user presses a command button (in GUI mode)

ii. The user presses a parameters button (in GUI mode)

iii. The user presses the STOP button (in GUI mode)

iv. A command is given from the commands file (BATCH mode)

v. A parameter is set from the commands file (BATCH mode)

vi. A STOP is given from the commands file (BATCH mode)

vii. A block of calibration data is received

viii. The calibration data stream is empty

ix. The calibration data originates an error

x. A block of control data is received

xi. The control data stream is empty

xii. The control data originates an error

xiii. A block of real data is received

xiv. The real data stream is empty

xv. The real data stream originates an error

xvi. A long set of calculations is going to start

xvii. A long set of calculations finished

We can see that the first set of three events and the second one are identical in the reaction of the system. This is clear from the point of view that only the *kernel* of the system will execute the reaction to such events, being the GUI and the Commands Interpreter simple interfaces to it. This is also true for several of the other points enumerated here. This shows us how our system can be modularized. I omitted here the events from the point of view of the system, i.e., the services the system gives to the user (for example, when the system shows some result from an analysis to the user in the form of an ALPHA plot).

### *3.1.2 Use Cases*

In this part of the design, we should clarify the functionality of the system from the user's point of view, i.e., what we want the system for. Here I will omit what the system is actually doing in each case. Some of the uses can be:

a) Get the significance and other statistics of a source from a set of observations in given periods.

b) Get information of how the real system (the telescope, in our case) is behaving (statistics on calibration and control data).

c) Produce nice (and fast, not necessarily so nice) printouts of some events and all the data.

d) Detect faulty real, control or calibration data sets.

Once we have listed all the *uses* of the system, we should try to figure out what the system must in fact do in order to fulfill each use.

| | |
|---|---|
| **Persistent objects** | Characteristics of reflector<br>Characteristics of camera<br>Characteristics of environment |
| **Session objects** | Date, time, ...<br>Type of analysis (source, laser, ... )<br>Results from calculations |
| **Concepts** | FADC entries (per channel data)<br>Real data<br>Laser data<br>Control data (LIDAR, Weather station, ... )<br>Logging information (warning, alarm, errors, ... )<br>Current logging data<br>Calibrated data |
| **Transactions** | Messages (events) |
| **Visual elements** | Menus<br>Command buttons, parameter buttons<br>Tool bar<br>Help handlers (status bar, tooltips, online help, what's this, task progress indicator, ... )<br>Visual representations handlers<br>Printouts handler |

Table 1: Objects identified in our analysis system.

### 3.2 Domain analysis phase

As I mentioned before, this phase is devoted to identify the different objects taking part in our system. In table 1 I give an example of some of the objects we can identify in our system.

This is of course not an exhaustive list of all the objects taking part in the system (it cannot be: only one person did it!). Nevertheless, I will try to outline the interdependencies of these objects in a global *class diagram* (see fig. 2). All these objects, from the operability point of view, can be grouped into *subsystems*. Each of these subsystems should be nearly independent from the others in their functionality, being the only relationship between them the exchange of information. In fig. 3 I outline the subsystems in which the system can be divided.

The main task in this step is to *construct and visualize all the subsystem's class diagrams* in our system. This really means to define the structure of the classes we are going to work with, with their members, methods, slots and signals (if any), all of them public, protected or private. This task is better performed in an "incremental" way (first defining the more important classes, even abstract ones, and their members and methods, and then fine-tuning them). But it is anyway a huge task, and my proposal is to write down first the conventions for the definition of the classes, members, methods, ... , and then to divide the job. On the contrary, if we just want to make a *brain-storming* one or more meetings, and state this structure once for all, I would suggest nevertheless to define first that list of conventions.

### 3.3 Define objects behaviour, Detailed design and Implementation

Specifying the list of classes that appear in our system, as well as their members and methods, together with their scope, is a big task, but not the biggest one. Perhaps a bigger effort comes in the description

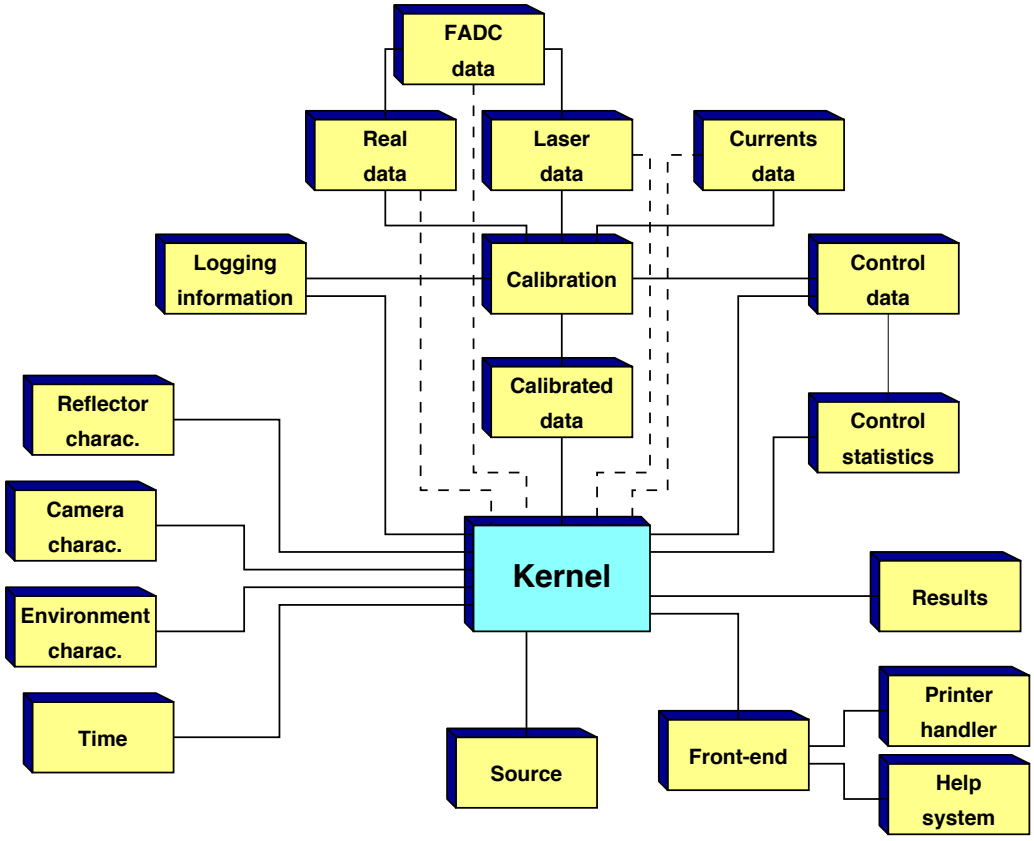


Figure 2: Outline of the analysis system's class diagram.

of *what each object is going to do in any possible context*. For every input the object could have, a clear description of how it is going to react must be done in order to understand and have a very clear view of what we have to implement in the final code. This is also a complex process, but just by doing the exercise of trying to understand what the object really does we will expend much less time afterwards in the implementation. In addition, this would the best source of documentation once the system is running, and will serve as the best introduction for new users of the software system, as well as for new contributors.

In the detailed design we will introduce platform dependent code as well as system and hardware specific code, if needed. This could be the case of differences in the GUI for different systems or differences in the flags for specific platforms[2].

### 3.4 Testing

Up to now I did not mention the fact that protection against defects and error correction can expend as much as 50% of the time and 50% of the cost of a project. Therefore, we must focus our attention in incremental software testing. This is the key point of the software methodology called *Clean-room Software Engineering*. The name is inspired in the clean rooms used in precision manufacturing, where

[2]Off-topic: nowadays this is usually not the case, and hopefully it will not be in our case, but we must be smart enough

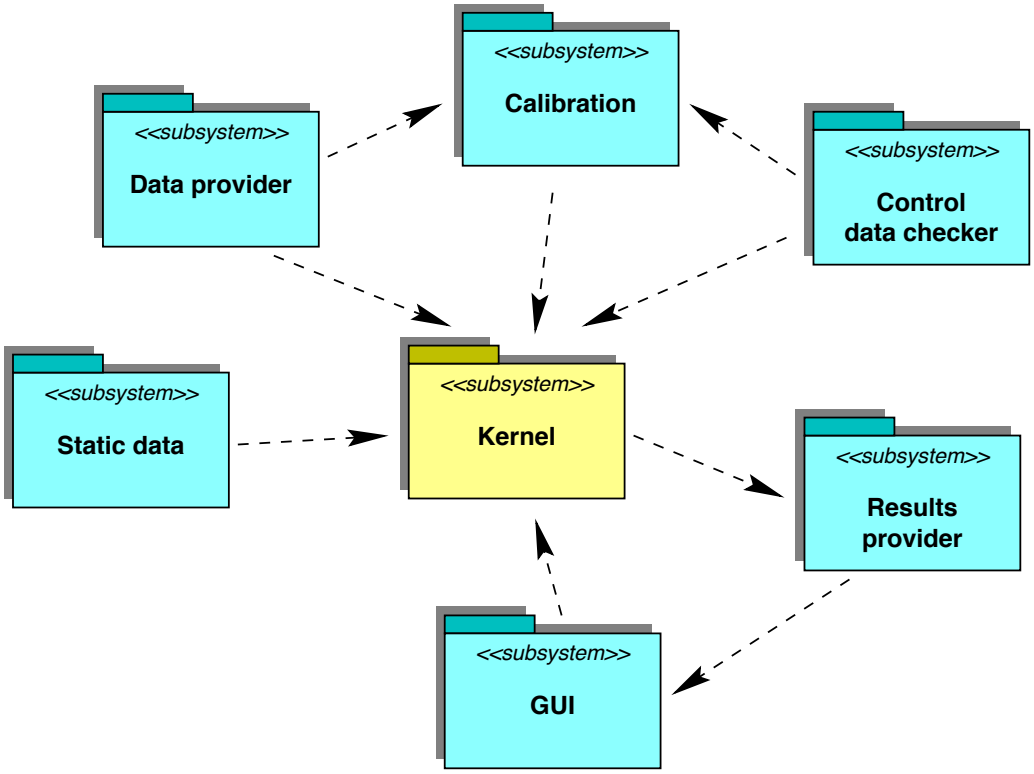


Figure 3: A view on the different subsystems in our analysis software.

statistical quality control techniques emphasize defect prevention over defect removal.

The clean-room approach is the so called *incremental pipe-line*. The whole process is divided in black boxes, which correctness must be separately proved. Once the whole chain of mini-systems are proven to be right in their output for all their possible inputs, each box is divided in smaller, detailed subsystems. Then we start again. This technique can lead to software that have remarkable high quality. However, the clean-room approach is a very technical one, which makes use of mathematical models and statistical techniques in order to reach his high quality at any given step.

I do not propose to use such techniques, since we are not prepared at all for such task. What I propose is an incremental testing of every sub-unit of our software, whenever a modification is made. Each of these sub-units would be separated from the whole system and separately tested with a whole set of different inputs. This will not only improve the software quality in terms of defect prevention, but also will enhance the modularity of our system.

In order to be able to do that, a clear set of rules for interfacing each sub-unit (say module, function, routine, whatever) with the rest of the system must be established. Of course, such set of rules should be approved at the very beginning of the development process[3].

[3]I have in mind to prepare an outline of such rules, together with some recommendations in the way of documenting our final code. They might be the subject for another document like this one in a near future, and a kind of seed for further discussion.

## 4 Documentation and conventions

From my personal point of view, there is something very clear: **having good documentation is extremely important.** I feel sometimes that having a simple system that at least works is what people think is sufficient. In my opinion, this way of thinking is our worst enemy. As far as we have time to do a good work (and we still have time) we must do it. Otherwise we will have problems sooner or later, and this problems can be hard to solve in a complex system like ours. In this context the documentation is never enough. I would propose an exhaustive documenting process in our development.

In addition, some conventions about the practical implementation (naming conventions, documentation of code, objects layering and inheritance, ... ) must be set up. In my opinion we have to be very strict in the sense of standardization. Perhaps there should be a responsible of the special task of taking care that standards and conventions are followed. Quoting Todd Hoff in his "C++ Coding Standard"[4] about "Standards Enforcement":

> "First, any serious concerns about the standard should be brought up and worked out within the group. Maybe the standard is not quite appropriate for your situation. It may have overlooked important issues or maybe someone in power vehemently disagrees with certain issues :-)
>
> In any case, once finalized hopefully people will play the adult and understand that this standard is reasonable, and has been found reasonable by many other programmers, and therefore is worthy of being followed even with personal reservations.
>
> Failing willing cooperation it can be made a requirement that this standard must be followed to pass a code inspection.
>
> Failing that the only solution is a massive tickling party on the offending party."

Of course, an standard is not needed for success. But it makes things easier and the development will go for sure smoother.

## 5 Conclusions

There are several conclusions about this document. I suggest using the development strategy outlined in section 2, namely using the steps *Requirements analysis phase*, *Domain analysis phase*, *Define objects behavior*, *Detailed design* and *Implementation*. In the process of design of our system, a good amount of documentation is generated, but in addition things will appear more and more clear. Together with this I suggest an incremental way of *testing*: in every phase of the development we should be able to check the design and detect faulty elements (this is specially true in the implementation phase). Finally, I propose the preparation of several documents about conventions and standards to be used in our analysis group, and which *should be strictly* followed by everybody.

[4] T. Hoff, "C++ Coding Standard", May 1999, `http://www.possibility.com/Cpp/CppCodingStandard.html`