Recent Developments in the Application of Object Oriented Technologies in the CERN PS Controls

M. Arruat, F. Di Maio, N. Gomez-Rojo, Y. Pujante
CERN, PS Division
CH-1211 Geneva 23, Switzerland

Abstract

The software architecture of the control system of the CERN PS complex [1][2] is strongly based on object concepts. Equipment modules, designed and implemented during the late 70s, introduced the concepts of abstraction and encapsulation, leading to an object oriented implementation during the 80s. The current software architecture, set-up during the past 6 years, implements a well-defined object model in the front-end computers from which the console-level classes have been derived. In this context, the integration of the object oriented technologies is a natural and continuous process.

This paper reports on the recent evolution of this architecture at the console level: the migration of the system libraries to C++, the introduction of object oriented Computer Aided Software Engineering (CASE) tools, the connection with CDEV [3] and the integration of Java.

1 Introduction

1.1 Objects in the PS Control

In the CERN-PS complex, control entities (e.g. device classes, device instances, front-end computers or hardware modules) are all described in a configuration database [4]. The control device objects are implemented in the front-end computers using the equipment module concept: every element belongs to a well defined class, which describes the structure of the instance variables as well as the Application Programming Interface (API), which is composed of properties. These objects are persistent, in the sense that they are created when the front-end starts and that their state is maintained by automatic back-up processes.

Application programs rely on workstation objects built from the configuration data and from the front-end's objects. The programmer is provided with the following libraries: (1) the "eqp" library (40 classes) that implements all equipment-related services, (2) the "ppm" library (10 classes) that implements access to the timing system, including synchronization facilities with machine events, and (3) the "err" library (3 classes), which handles exceptions and error logging.

1.2 C++ Migration

A migration from C to C++ has been executed for these workstation libraries, including re-coding the more important one: "eqp". The motivation was to organize in a better way the long-term maintenance of these ever changing software packages, in the context of a high turnover of the developers. We aim at having a more robust code and maintenance activities both more interesting and more efficient.

As concrete objectives: the life-time of the operators' programs required improvements, unstable implementations of the synchronisation services required some encapsulation, a revision of our equipment access library had to be produced and we have to be prepared to continuously cope with new requests.

2 Console Objects Design

The objects described in this paper are used in the software developed for the operator's workstations (Unix). Some particular aspects of these console objects are described below.

2.1 Equipment Objects

Equipment objects represent a homogeneous part of the equipment (e.g. a list of power-supplies) and implement the transaction functions (read/write/call). A simplified class diagram is shown in Fig.1. EqpElement objects describe individual pieces of equipment, each of which belongs to one EqpModule object, which implements their front-end level class.

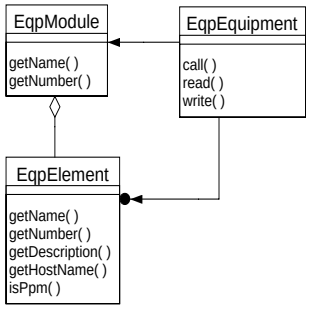

Fig. 1: Simplified model of the Equipment class

2.2 Static Objects

Many console objects belong to a special category called static. These objects are built from the configuration database (Oracle tables). Their instance variables are defined by the configuration database and cannot be modified directly by the application programs. As examples, the EqpElement and EqpModule classes, displayed in Fig.1 implement static objects. These objects are built through a 2 step process: (1) data structures are extracted from Oracle tables and distributed into Unix files and (2) C++ objects are created from these files when required and kept in the task's address space afterwards. Such a process is very close to the behavior of an object oriented database. This two step procedure also fulfills the requirement of having one independent sub-net per accelerator by distributing redundant and read-only copies of the configuration data to many servers. Static objects that don't have these requirements are implemented by direct connections between C++ methods and PL/SQL (the database programming language) functions.

2.3 Meta data

Another aspect of the console objects is that the description of the front-end classes is accessible through C++ classes and that this description is complete enough to allow a generic implementation of many operator facilities: displays, control panels, archives, etc.

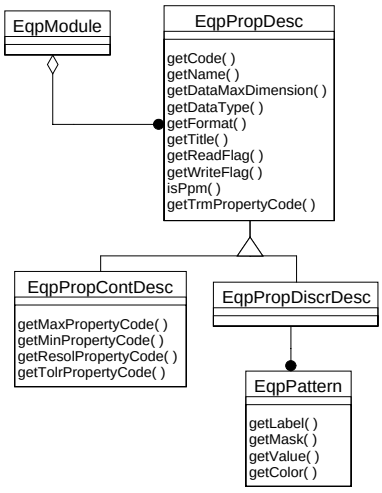

Fig. 2: Property description classes

As an example, Fig. 2 illustrates a part of the description of a front-end class. This description is composed of a list of properties supported by the equipment-module. Property descriptions (PropDesc) are provided, with the distinction beween continuous (PropContDesc) and discrete (PropDiscrDesc) properties, for which the data (bit patterns or enumeration) belong to a closed set, described by means of patterns.

3 Object Oriented Software Engineering

When trying to use software engineering methods in the software design phase, one has to cope with two consecutive problems: (1) to build competence in a method and in the correlated tools and (2) to be able to maintain up-to-date models all along the software exploitation phase. This second problem is the more difficult to tackle: although, in theory, models should be kept up-to-date, in practice, once people move on to coding, models are not edited anymore. As a result, while the initial design is neither complete nor exempt from errors, the distance between the code and its model grows with time.

Compared to Structured Analysis and Structured Design (SASD) methods, object oriented methods offer new possibilities: because of the strong coupling between the design models and the code, one can try to use tools for keeping models and code synchronized. An attempt was made to follow this line, with some success.

3.1 Reverse engineering

The "eqp" library, which was a C library, was first manually reverse engineered into an OMT [5] class model, reflecting exactly the structure of the C objects. This was very useful for the maintenance of the C code because the class model is a good communication media. Such a procedure, however, was not efficient enough to prepare important changes, because of the duplicated maintenance of both the model and the code.

Another approach has been explored with the "ppm" library, which was already a C++ library. Using a commercial product (Rational Rose®), the C++ header files were analyzed in order to produce class models in an automatic manner. This has been a rather tedious process, requiring a lot of interaction as well as some help from the company (consultant). It took us 2-3 months of an expert engineer to master the technique. Nevertheless, the results are satisfying enough to adopt the procedure of using reverse engineering tools to re-generate models from code when required.

3.2 Code generation

The new C++ version of the "eqp" library was produced, with the following steps: (1) the new version was re-designed by means of OMT models, (2) it was implemented using the code from the C version, (3) the code was analyzed, in order to re-generate an up-to-date

model and (4) diagrams (class and scenario) have been manually produced as a design documentation.

A more systematic approach was applied for the new version of the "ppm" library: (1) the design was made by means of OMT models, using the analyzed classes from the previous version (2) the model was used to generate code (headers and functions skeleton), (3) the new version of the library was implemented from this code and from the previous version. The model has been kept up-to-date by adopting the method of always updating the model first.

3.3 Consequences on the code

There have been a lot of consequences on the code, especially on header files. Code analyzers introduce new constraints that compilers, even C++ level, do not require. It was, for instance, necessary to get rid of all circular dependencies by replacing some inclusions of class definition with forward declarations.

Code-cycled files (the ones that can be modified from the model) include a lot of the tool's specific annotations, that identify the segments of code produced by the tool and the segments of user code. These annotations allow, for instance, to preserve the methods' bodies when re-generating the code from the model but they also reduce the readability of the code files.

3.4 Documentation

An additional benefit obtained in introducing CASE tools was the production of on-line documentation. We are not able to maintain man pages, or any equivalent on-line reference documentation if not generating them in an automatic manner from code files. In this perspective, Web pages are now produced from the C++ header files to describe the classes, their syntax and some description. This has been achieved by adopting some conventions on the description entered into the models and by using converters which transform these into HTML format. These converters are homemade filters that convert Rose annotations into doc++ comments [6]. This proves to be a practical method of producing on-line reference documentation from header files. This procedure has been adopted for most of our C/C++ libraries, including some front-end ones. This method also exists in the Java environment (javadoc).

4. Programming Issues

4.1 Container classes

In the design of the class libraries, we restricted the usage of container classes to the following categories: unbounded ordered lists and unbounded key sets. In addition, only pointers to object are stored into containers, instead of objects themselves; this implies explicit destruction but reduces the usage of copy constructors. Unlike Java, such container classes are not part of the C++ default environment but they have been standardized later in the Standard Template Library (STL). For now, we uses IBM's container classes, which are also based on templates and iterators.

When using class templates, we encountered the problem of having some template classes generated twice. This was fixed by including all template classes into the libraries and by preventing the client application to generate them, via a compiler option.

4.2 Exceptions

Using exceptions is another important choice to make. It was decided to use them extensively. We had good experience with this choice: incomplete error treatment was detected quickly and easily solved with debugging tools because the program breaks into the debugger where the error is detected and not on secondary effects. For example, raising an exception is safer than returning a NULL pointer as an error indication. In addition, exceptions are the simplest way to cope with errors within constructors.

Another very useful aspect of the exceptions is that they provide a more efficient way of logging errors. The previous convention was to log errors where they were detected using dedicated functions and then to return an error indication for the caller, which can, as well, log a higher level error. We replaced this with a hierarchy of error classes which provide logging functions. Objects thrown as exceptions always belong to this hierarchy. This allows a better filtering of the error logs and the composition of more complete error messages. It has some similarities with the Exception class of Java, the base definition of which being to store an error message.

4.3 Data Objects

While the libraries are strongly oriented towards generic software, the data exchanges between the application programs and the front-end objects require dedicated objects that encapsulate the actual data representation and provide the various conversion and buffer management services. Dedicated data objects were developed for this purpose. The code duplications and the memory leaks were significantly reduced this way. The data classes are implemented by means of class templates in order to support many data types.

Another role of the data objects is to be a data exchange medium between different libraries, like transmitting composite data read from the equipment to some display services. In this role, some more widely used classes (e.g. cdevData) would probably be more adequate than a local implementation.

5. Connection with CDEV

The connection of the CERN-PS equipment classes to TJNAF's CDEV [3] has been prototyped, using the CDEV's method of interfacing control systems. This method, based on deriving local classes from cdevService and cdevRequestObject classes, is rather simple, especially for control systems using a restricted API, like the CERN-PS one.

A preliminary version of this connection was implemented with CDEV 1.3. The supported messages are restricted to "get" and "set" verbs + attribute. The mapping from CDEV concepts to CERN-PS ones is the following: device names are element names, device classes are equipment modules and attributes are properties. The implementation took about one month and one thousand lines of code. Error reporting (a general feature) and ppm integration (a CERN-PS feature: pulse to pulse modulation) will require additional work for a complete implementation.

This connection provides the possibility to use some of our software packages outside of the CERN-PS context. The work done with CDEV was also very valuable in the re-design of our classes: we adopted some CDEV concepts, like the data objects and the "Service" class.

6. Java

It is being considered to produce some application programs in Java, especially in the perspective of a new machine (Antiproton Decelerator). As a result, a Java implementation of the console objects is being developed.

The direct connection between Java classes and the C++ classes through JNI (Java Native Interface) was prototyped. This solution will be adopted for providing the programmers with a Java API to the CERN-PS controls. It will only allow running Java programs on the control's workstations, but it will be the base for launching developments with Java.

The Remote Method Invocation (RMI) has been prototyped as well, this solution is considered for implementing a Java API for remote clients.

We have now to adopt an API that not only can be easily implemented by means of wrapper Java classes talking to C++ objects via JNI but that can also be implemented with remote objects, via RMI or CORBA.

Many application programs are still produced by C/Motif programmers who don't use C++ or OO design. In this case, Java, instead of C++/Motif, is being considered as the OO learning environment.

7. Conclusions

The migration of the equipment access library from C to C++ was the central part of an important evolution toward OO technologies in the domain of the software design and implementation. Many positive consequences of this evolution can be observed.

Firstly, the quality of the code has been greatly improved, in terms of robustness and maintainability. Memory leaks are no longer disturbing the operators' environment and we could easily cope with the required extensions during 1997.

Secondly, the introduction of a more routine usage of CASE tools improved a lot our working methods: better team work, more care in the design and a better design documentation. As an example, during the last year, each of the authors of this paper provided a significant extension of the "eqp" library.

Thirdly, replacing C with C++, as a programming language, not only goes with the interests of professional programmers, but also allows us to get rid of local conventions for coding objects in C. This facilitates the connection with another OO equipment access API and the extension towards Java.

After to the migration of the system libraries to C++, the next step, in the application of OO technologies in the CERN-PS context, will probably be the application programmers starting to use Java.

8. Acknowledgments

We benefit from the services of our CERN colleagues (ECP/IPT group) who take care of the commercial and technical aspects for making CASE tools available.

The French office of Rational Software Corporation offered some very helpful assistance in the reverse-engineering phase.

CDEV from TJNAF is a very useful reference of an object oriented equipment access for the concepts and their implementation.

9 References

[1] F. Perriollat, C. Serre, "The new CERN PS Control System - Overview and Status", in ICALEPCS 1993 proceedings, Berlin, Germany, Nucl. Inst. and Meth. A352 (1994).

[2] CERN PS Controls, "PS/CO Group" , http://www.cern.ch/CERN/Divisions/PS/CO.

[3] C. Watson, J. Chen, D. Wu, W. Akers, "cdev", http://www.jlab.org/cdev.

[4] J.H. Cuperus, M. Lelaizant, "Integration of a relational database in the CERN/PS Control System", this conference.

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object-Oriented Modelling and Design", Prentice Hall, 1991.

[6] M. Zockler, R. Wunderline, "DOC++", http://www.zib.de/Visual/software/doc++.