

MOOSE : SOFTWARE DEVELOPMENT FOR LHC

C. ARNAULT

LAL Orsay - arnault@lal.in2p3.fr

MOOSE (CERN-RD41) COLLABORATION

The CERN's R&D project MOOSE (RD41) has been formed to study the technologies, the methodologies and the human behaviors that could help provide solutions to the problems that HEP software environments must address in the next few years, in the context of analysis and reconstruction software for the LHC experiments. During this project, the cultural and technological aspects are considered as key issues as well as projects constraints and requirements. The main problems addressed so far by this project (mainly the team and group behavior in the cooperative actions of the software life-cycle and the impact of software engineering techniques on our way-of-life) will be presented and discussed.

1 Introduction

The MOOSE [CERN-RD41] project was formed to work on and establish a well structured software development environment for LHC offline software, adapt the use of OO methods to the HEP environment and eventually recommend the OO approach together with the guide-lines for using it to our community.

The need for such a project is enforced by a set of observations on the recent history of the computing environment and the specific problems raised by the LHC era, such as :

- how to manage the wide geographical dispersion of software providers and users,
- how to give users access to increasingly sophisticated software while hiding its complexity
- how to incorporate the turn-over of technology into the long life time of the LHC software.

The R&D project MOOSE [CERN-RD41] ¹ was approved in June 1994 by CERN's Detector R&D Committee (DRDC), and is currently being over seen by the LHC Computing Review Board (LCRB). The final goal is to give conclusions on what should be recommended at the end of 1996.

In the original proposal four areas of research are mentioned : methodology with special attention given to the object orientation, tools, implementation languages and persistence. A well defined context for designing and exercising prototypes, the reconstruction software for the two LHC experiments Atlas and CMS was exploited to gain experience on methods and other aspects of software engineering such as requirement specification or management and process control. Now we are coming to the stage where we have a working prototype.

We have come to realize that there are management issues which must be addressed for success. It is felt by the collaboration that one of the major achievements has been also a study of collaborative software development by a group of highly dispersed and different individuals.

This presentation will discuss some of the topics that have been addressed so far by this R&D project after one and a half years of existence.

2 Project organization

2.1 *Prototype studies*

The project activity is organized around pilot projects (quite similar for the Atlas side and the CMS one) for reconstructing electrons in the EM Calorimeter and Inner Tracker. The basic steps consist typically of re-engineering a piece of existing software (or strategy or algorithm) by :

- extracting the overall strategy from the domain experts
- developing a model
- implementing code which could read simulated data.

while trying several different technologies and methods.

2.2 *Group dynamics*

Although the total number of people participating the actual day-to-day work was rather small (around 10 to 15) the team was spread among several institutes in Europe which require us to keep a good level of organization in every step of the software life cycle. Therefore this was a quite good experimental condition for this study.

The typical relationships established between the members of the project consisted in regular meetings - every six weeks - with restricted work sessions (two or three persons) in between. The main difficulty in this kind of organization is to share the information in order to minimize the duplication of efforts. Therefore we put emphasis on mechanisms, tools and conventions that help sharing actions or informations such as :

- an extensive use of WWW based hyper-text documents. The complete project status (definition of the project, areas of research, minutes of the meetings, etc...) is maintained within a set of WWW documents which extends over the various sites of the project members.
- problem analysis sessions during the workshops, resulting in specifications in informal natural language or graphical representations of the concepts (classes, operations, etc...) using either formal or informal notations.

- code review sessions where a few reviewers evaluate the quality of the code of one particular isolated development, using conventional criteria (such as understandability, completeness, choice on variable names etc...); the result of this evaluation is summarized and interpreted by a moderator who proposes a set of actions to be taken by the author of the code. This kind of session turned out to be extremely useful for the community to understand a piece of the software and to participate its life cycle.

2.3 *Software Organization and Configuration Management*

Managing a complex project requires that the logical and functional description of the problem through domains and packages is mirrored by the software packaging and the disk organization. This is also a natural consequence of the specification and analysis phases since the *responsibility* for the various components of the developments was always assigned to a particular subgroup. This sort of question is not extensively addressed by traditional OO methods but we realized that a well structured approach is needed for a practical management, especially when the project gets a significant size.

The physical management (in terms of disk environment) is based on AFS in order to share the physical components of the team production. Responsibility on various software productions has been assigned to a few members of the project who take care of the security control with the help of access control mechanisms that AFS provides.

CVS was used as the basic version control tool and we studied various correlations between this tool and other questions, such as sharing the repository between the users, how to reflect the logical and structural division in subsystems derived from the early stages of the analysis or how to exploit the CASE tools in this framework. Comparisons with a few similar products show that CVS remains a quite valuable solution for this problem.

We are now studying the basis for configuration management that describes the dependencies between packages (sub-projects) involved in the development. This fairly new notion should provide means of controlling the *evolutions* of each particular subset of the project, taking into account the propagation of changes. Unfortunately tools for managing such features don't really exist yet.

3 **Methodology : Requirements, analysis and design**

Software engineering methods are meant to organize the process which provides a computer representation of a particular problem. They are generally based on steps such as : a complete and precise understanding of the problem domain which usually yields a set of *requirement specifications*, the analysis of these specification provides a set of concepts which represent *abstractions* for the problem and finally the design translates these abstraction into computer oriented components.

The process of going from getting user requirements, finding the abstractions, designing, coding, testing and using and then going round again with new requirements or maybe a revised analysis or design is known as the software life cycle.

Note that the cycle may loop back to any point even to the beginning.

However, the way one understands the process and drive the cycle leaves a great degree of freedom and for instance in MOOSE we had two approaches : in the Atlas side, the strategy reflects closely the hardware design, whereas in the CMS side the analysis allows several strategy for class organization.

3.1 Problem Specification

In general the first step turned out to be extremely rich : in order to obtain the precise and complete problem statement, we had to extract the deep knowledge on data structures and algorithms from our experts in the reconstruction domain. The algorithms used in the reconstruction studies had to be fully understood, every name such as a *cluster*, a *track segment*, a *particle candidate* had to be completely defined, the roles of all these concepts established, the relationships between them identified. We were able for instance to identify the need for persistence for some of the objects involved and the organization of objects that must both reflect their geometrical characteristics and optimize the algorithms used to search the particle paths.

3.2 Analysis

This process of finding the concepts and their properties (which is the abstraction process) occured in work sessions where the existing software (typically written in Fortran) was re-analyzed. The isolated concepts such as a *tree-oriented* structure used to hold elements found in a search path during the reconstruction were then worked through as autonomous developments.

The analysis sessions showed us for instance that the geometry model needed in the reconstruction is not the same as the one provided in the simulation applications. Therefore a new set of abstractions such as the *layers* that handle the logical sub-structures of the tracker that are able to provide *space-points* is introduced. This also generates the need for specifying interfaces between models (and between the corresponding designs and implementations).

3.3 Design

The most important concepts that we exploited were the object interaction graphs (OIG) typically described in the Fusion method and the notion of *design pattern* described in ².

An object interaction graph shows how a specific scenario is carried out by a group of collaborating objects. It describes the object that drives it and the interactions with other objects that may occur during this operation. This kind of analysis yields a quite coherent and global vision of a system.

Design patterns are defined as a description of a recurrent problem and of the core of the possible solutions to it. Hints for technical implementation are often provided. Typical examples of such problems are the notion of *factory* mechanisms used to delegate and manage the creation of objects, *collection* managers and *behavioral* patterns that encapsulate typical behaviors such as how to iterate in a

collection or select a strategy. Design patterns represent a terminology, a taxonomy of features that are quite common in any kind of development and that can find a solution independently of the implementation language.

4 Tools

While the expressiveness of the various formalisms was found to be quite good, the full power of these methods was reached only when a tool supporting interactive manipulation of the models as well as automatic code generation was available.

The tools are an essential element for a successful use of a method, especially for keeping up-to-date the code relative to the successive evolutions of the model.

However the commercial issues are essential : the price per license remains high even for a wide distribution, and on another hand identifying the clients (the developers) of such tools is not entirely understood : whether they should be made available to anybody in the community willing to participate in the development or just reserved to a kernel of experts is not clear yet. The cultural and technical issues must still be evaluated in more details.

Clearly one cannot envisage today to select one *good* candidate for such a tool that would address the needs of all the HEP developers and would survive the life time of the experiments. Therefore one must identify some criteria that will ensure a portability of the designs between tools (and methods). Among these criteria having a textual representation of the model appears to be critical : this representation can be seen as a specification for the problem and as soon as the concepts involved in the model (the grammar of the method) are well identified, converters may generally be built that would translate one language into the other.

We have gained experience with both commercial tools (Objecteering from Softeam and Rose from Rational) and a home made tool (OMO) that exploits a formalism partly derived from the Fusion method.

- Objecteering and Rose :

Their philosophy is quite similar : A graphical editor helps the designer to enter the elements of the model using the formalism offered by the method, and a textual representation of the model is produced in order to act as the main repository of the project (as opposed to tools that maintain the project components in a specialized database management system). Then services are provided in the tool for deducing from this textual representation various other forms such as code in a given programming language (typically C++ but others are possible), the documentation at different levels or even the complete environment for building and managing the target applications.

The textual representations of the model they offer are designed to be rather independent of the targeted programming languages (although the influence of C++ cannot be ignored!).

Objecteering supports a proprietary method (Class-relation ³) that provides models for structural, dynamic and operational aspects whereas Rose supports the method proposed by G. Booch.

- OMO :

This Object Modeling tool (Omo¹⁰) was developed within the MOOSE project and lies on principles that differ slightly from Objecteering and Rose by the fact that it is explicitly built for Eiffel development. Here the textual representation of the model is mostly the Eiffel source files. However, a support for state transition diagrams and OIGs is provided and turned out to be quite helpful.

Omo was designed as a lightweight tool to help in the generation and understanding of code rather than being the starting point for the design.

By building our own tool we have been able to try variants of the notation and of the model while being cost-effective since it is built with Tk/tcl⁴ and Eiffel.

5 Implementation Languages

MOOSE decided originally two approaches for implementation languages. The CMS tracker reconstruction package is being written in C++ whereas the Atlas group developed its first prototypes using the Eiffel language.

Eiffel is easy to understand, to learn and to operate. Moreover, most of the team members had little experience with OO programming and Eiffel was an efficient path from OO concepts to implementation.

The constraints from industry (the vast majority of commercial - or public domain - products are targeted at C++ environments) make life difficult for *non-standard* environments. Therefore moving to C++ is a likely evolution not only for the ATLAS prototype but for all software production for the next few years.

What is true for the next five years may not stand for the lifetime of the HEP experiments, nor match the essential requirements for integrating the existing software (based on Fortran). Thus the most important issue when choosing a language is its capability of being interfaced to other environments. Fortunately, the introduction of open systems (such as Unix) is a great help in this area and we know how to interface most of the currently used languages (C, C++, Fortran, Eiffel).

Implementation languages should become less and less important in the future. This means that project managers should feel free to select one language or another. But the key issue will be more and more to provide to the clients a portable way of understanding the visible data sets or to interface the services in any environment.

Thus in opposition to the constraint given to the developer which is mainly the availability of general purpose libraries or the ease of producing new ones, the constraint given to the project manager will be to describe the interfaces (data sets and services) using standard specification languages.

6 Conclusions and Outlook

The preliminary results that emerge from this first year's study show that even more than the concept of object orientation (which nevertheless is the most mature and coherent approach in the domain of SE methods) the demand for an engineering

approach to software production seems to be the principle criterion for a quality improvement process. In particular, we have shown and experienced that the following actions are essential in this process :

- establishing a substantial training and teaching program in our community.
- Widen the cooperation throughout the software life cycle from domain experts to software engineers and from the analysis of the problem to the design and code reviews.
- introducing explicit quality criteria that can be tested upon at every stage of the software life cycle, through conventions, code reviews or management tools,

We now start understanding what could be the paths for solving the technological difficulties that remain. Most of them deal with information or software production sharing or with project management in general. These questions are critical generally when different technical choices are envisaged by different sub-groups, such as implementation languages, CASE tools or object definitions.

Clearly a uniform solution is not affordable, not only because of the sociology of the HEP community but also it is in likely contradiction with the principle of encapsulation or independence that we would like to apply to any component of a project. A few key ideas are now emerging (although they are not entirely new for the computer science community) and will likely play a major roles in our next studies :

- interfaces :

In order to advertise the services offered by a software component one must very precisely specify its interface, showing what is the information that may be accessed, which operations can be performed and the syntax of the protocol when one uses these services.

Thinking in terms of interfaces is a design strategy since it may be applied in any implementation language and at any granularity level.

- specification languages :

Using a language to specify a problem consists of using a formalism (a syntax) for describing the specifications without worrying about the implementation details. Many such languages exist today that range from data definition (known as DDL) to algorithmic languages (such as Z or VDM).

Besides offering a means for specifying the interfaces, this technique will significantly limit the impact of the traditional war between languages or methods since several international standards are now arriving from industry to cover specialized domains such as the data bases (with ODL proposed by ODMG) or physical object definition (with Step/Express proposed by ISO). We can reasonably predict that these standards will be a central communication point between software components but also between designs. A recent attempt to specify our pilot project using Z shown that it clarifies one's thought about the problem.

- patterns and frameworks :

Organizing the software architecture around the notion of a framework of reusable components that are responsible for providing services to each other is a direct consequence of the basic principles of the OO methods. The recent developments on design patterns that give a formalism to the construction of frameworks have already influenced the way we have achieved the design of the MOOSE prototypes. The well known problem of building class libraries will surely benefit largely from this new way of thinking.

References

1. The complete information about authors, project definition and current status can be found on the Web at http://www.cern.ch/OORD/Home_oord.html.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley 1994.
3. P. Desfray, *Object Engineering - The Fourth Dimension*, Addison-Wesley 1994.
4. J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
5. G. Booch, *Object Oriented Analysis and Design with Applications*, Benjamin/Cummings 1994.
6. B. Meyer, *Object Oriented Software Construction*, Prentice Hall 1988.
7. T. Atwood, J. Dubl, G. Ferran, M. Loomis, D. Wade, *Object Database Standard : ODMG-93*, R.G.G. Catell - Morgan Kaufmann Publishers 1993.
8. ISO 10303 Industrial Automation Systems and Integration - Product Data Representation and Exchange, ISO TC 184/SC4, 1992.
9. R. Cailleau et al. *The use of the World-Wide Web in HEP*, CHEP-94 proceedings.
10. S.M. Fisher, D.J. Candlin, *Omo - A Tk/tcl based object modelling tool to support Eiffel*, CHEP-94 proceedings.
11. E.H. Durr, A. Duursma, N. Plat, *VDM++ language reference manual*, Afrodite report, 1994.
12. DICE-95 and Age, CERN Atlas notes.
13. S.M. Fisher, P. Palazzi *The ADAMO System Programmers manual*, CERN ECP and RAL.