STORAGE AND SOFTWARE FOR DATA ANALYSIS

R. Brun[1], R.G.Jacobsen[2], R.Jones[1], J. Moscicki[1], M.Nowak[1], A.Pfeiffer[1], F.Rademakers[3]
1) CERN, Geneva, Switzerland
2) University of California, Berkeley, USA
3) GSI, Darmstadt, Germany

Abstract
This track combined exposure to the software technologies and packages relevant for LHC experiments and the engineering aspects of software development. The lectures provided an overview of LHC++/Anaphe and ROOT and covered those aspects of software engineering most relevant for HEP software development. It showed, in a practical sense, how software engineering can help in the development of HEP applications based on the LHC++/Anaphe and ROOT software suites and also gave a taste of working on large software projects that are typical of LHC experiments. A series of hands-on tutorials performed by the students were based exercises to solve given problems. The tutorials followed the natural progression of physics analysis exploring the major packages of LHC++/Anaphe and ROOT on the way.
This paper presents an overview of the software engineering lectures by R.Jones, the tutorials and feedback session. Details of the lectures on LHC++/Anaphe and ROOT are reported in separate papers of these proceedings.

1. INTRODUCTION TO SOFTWARE ENGINEERING

A definition of what is meant by software engineering gave a starting point for this lecture which then went on to explain how the scale of the software project determines the software process required to successfully run the project to completion. Developing a model for a large scale software system prior to its construction or extension is as essential as having a blue-print for constructing a building. Good models are necessary for communication between the project stakeholders (members of the development team, users and management) and to assure architectural soundness. As the complexity of the software system under development increases so does the importance of good modelling techniques.

Various processes exist for object-oritened (OO) software (OOIE, OMT, Booch, Fusion, Syntropy, OOSE, Unified etc.) and have varying definitions for the phases involved during the project. The history of these OO software development proceses was described and how this lead to the appearance of the Unified Software Development Process (USDP) as a defacto market standard taking elements from previous development processes. The authors of USDP recognised the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance. USDP is centred around the architecture of the software under development and follows a number of iterations driven by use-cases. Each iteration can be treated as a mini-project resulting in a new release of the software and following all the phases of the software process.

The importance of a notation to document and visualize the models developed as artifacts of the process phases was explored and the structure of the Unified Modelling Language (UML) notation

was shown. The UML represents a collection of best engineering practices that have proven successful in the modelling of large and complex systems. USDP incorporates the UML as a standard into its development processes and products, which cover disciplines such as requirements management, analysis & design, programming and testing. The UML can be used on varying software development projects since it is independent of the process (i.e. USDP) being followed. An overview of the UML starting with use-cases showed the basic structure of the UML, the notation and types of diagrams that can be used to describe the software under development. Emphasis was put on use cases as a means of driving the development and how they are used to capture the requirements. The essential purpose of the requirements gathering is to aim development towards the right system. This is achieved by describing the requirements (i.e. the conditions or capabilities to which the system must conform) well enough so that an agreement can be reached between the stakeholders on what the system should and should not do. The issues involved in ranking use-cases to determine their priority and establishing how they map onto the project iterations was described.

Once the use-cases have been explored and ranked attention can move onto establishing the basic architecture of the software starting with deployment diagrams showing the nodes (processors) and interconnections (networks) that represent the environment in which the software will run. This environment is populated with the various modules of the software identifying interfaces to legacy systems (hardware and software) and the relationships between components. Details of the software are omitted from such diagrams to allow the basic disposition and architecture to appear.

2. SOFTWARE DESIGN

This lecture carried on the description of architecture and showed how work progresses towards establishing a more detailed design. The task of design was introduced as consisting of three levels: architecture, mechanistic and detailed based on the scope of the decisions made. Each level was further defined to show it's goals, techniques and deliverables. At the archiecture level the emphasis is on identifying the major software structures such as subsystems, packages and their interconnections. At this point it is possible divide the work into smaller tasks based on domains or subsystems that can be developed by different teams or individuals possibly in parallel. The desirable qualities of archiectural design was listed including layering sub-sytems to reduce coupling and promote independence. Other qualities such as well-designed interfaces and scalability were also mentioned but the most important quality of any design is that it is easily understandable.

At the mechanistic level attention moves to establishing the relationships between groups of classes (i.e. the mechanisms by which they are linked) and at the detailed level the developer examines the internal structure of individual classes by identifying necessary attributes and methods. The transition to implementation is made selecting the appropriate mapping for aspects such as associations and operations taking into account deployment (e.g. the association between two classes may be different if they are on separate machines or in separate threads), code ownership and practicalities linked to the use of underlying software packages. The UML class, sequence and collaboration diagrams were explained and examples drawn from the exercises. As design progresses through the three levels, more detail is added to existing diagrams (e.g. adding methods and attributes to classes) and new ones are drawn.

The concept of patterns that give examples of how several classes can work together in a given domain to address some problems was introduced. The ability to profit from design approaches used in other projects was discussed as a means of supplimenting the developers' knowledge and experience. Examples of how to apply patterns to analysis and design taken from LHC++/Anaphe and ROOT were given. While patterns are rightly considered as an excellent aid during analysis and design, the audience were warned about their abuse and some of their short-comings.

Finally an appreciation of the advantages and problems of UML were described as well as future likely changes proposed for inclusion in the release 2.0 of the standard scheduled for 2001.

3. SOFTWARE TESTING

This lecture covered the basic principles of software testing and why programs have defects.

The goal of software testing is to ensure the software under development has sufficient quality by evaluating the artifacts (documents, diagrams, source code etc.) of a project. The intention is to find defects (bugs) so that they can be removed and demonstrate that the software does meet its specifications. As a side effect, it builds confidence in the project members that the software is ready for use.

Testing includes both validation and verification of software. Verification implies executing an implementation of the software and has traditionally be the focus of software testing in HEP. This implies that no software testing is performed before an implementation is ready (i.e. once the requirements, design and implementation phases have been completed). A complimentary activity is that of validating the artifacts at each stage of the software process. For example, the design diagrams can be validated to ensure they satisfy all the requirements. The requirements document can be validated to ensure it is consisitent, complete and feasible.

Once an implementation exists, it can be validated against the design diagrams to ensure it faithfully realises the design and exhibits general design qualities (listed above). Validation has been shown to be more effficient that verification because each artifact of the software process is checked as it is produced and so errors are caught earlier when they cost less to correct.

The different phases of software validation and verification were enumerated and described:

Unit tests – tests peformed on individual classes;

Integration – tests performed on several classes, components or sub-systems to validate their interfaces;

Regression – repeating previously executed tests after a modification has been made to ensure the defect has been removed and no new ones have been added;

Acceptance – final testing performed at the user's site with their data.

Different techniques can be applied for validation and verification. Reviews and inspections are the most appropriate manner of validating documents, diagrams and source code. Three types of reviews were listed:

A personal review where an individual developer examines their own artifact individually;

A work-through where a developer presents their artifact to her/his fellow developers (peers) who are asked to comment and make suggestions;

An inspection which is a structured review involving an inspection leader, the author(s) and a number of peer reviewers. A number of meeting are held to inspect the artifact and all issues are logged and followed-up by the authors(s).

A number of elements need to be put in place before a review can take place:

A checklist of the most likely errors must be available to drive the process and indicate to the reviewers what they should be looking for;

The project stakeholders must accept that a review will "front-load" the cost of a project since the review will take some effort (even if it is only a couple of hours) and that the development

cannot proceed to the next phase until it is completed while (hopefully) the total amount of testing required will be reduced;

The review should be seen as a phase of the software process and not a personal assessment of the authors;

Training will be required for inspection leaders if the process is to run smoothly to a successful completion in a reasonable amount of time.

As an example, guidelines were shown for inspecting C++ sources code. Much of the work involved in inspecting source code can be automated using a coding rule checking tool to parses source code files and produce an evaluation report identifying which guidelines were violated. This report can then used to make modifications to the source code where necessary but violations of the guidelines should be accepted where they can be justified.

Techniques for validation were also addressed including black-box and white-box testing. In black-box testing the specifications and interfaces of the software is validated without examination of the structure of the source code itself. While in white-box testing, knowledge of the internal structure of the software is used to develop suitable test-cases. Boundary-value testing (e.g. varying the input parameters to object methods) was given as an example of black-box testing.

Coverage and path testing where knowledge of the internal structure of the software is used to develop test-cases that execute as many paths through the code as possible were given as examples of white-box testing.

Again, CASE tools can help to automate important fractions of the work involved in black and white box testing. The example of the Insure++ tool which can be used for code coverage of C code was given. Similarly tools exist for static source code analysis (e.g. LINT, Logiscope), memory leak checkers (e.g. Purify, Insure++) and boundary checks (e.g. "T" testing tool). Performance of the software and identification of bottlenecks can be identified by compiler profilers and scripting languages such as Expect are very useful for writing tests-cases.

Following the assumption that prevention is better than cure, a number of programming techniques available in many high-level languages were listed as being particularly error prone and difficult to master. The list includes dynamic memory allocation, parallelism, recursion, floating point numbers, interrupts and pointers. Unfortunately, these programming techniques are amongst those that improve the performance or efficiency of software and since HEP software is often at the limit of what is possible many of these techniques and often necessary.

The lecture concluded with a set of axioms about testing that can improve the way most software developers approach the subject and by trying to answer the most difficult questions concerning software testing:

How much testing is enough?

When or why should we stop testing?

When is the software ready to be released?

Since no software can be fully tested the concept of risk-based testing as a means of prioritising the choice of test-cases to be made was suggested. With such an approach, test-cases are developed to verify that the most important risks have been addressed while testing on low-priority aspects of the software may be dropped based on general agreement between stakeholders of the project.

4. LONG-TERM ISSUES OF SOFTWARE

This aim of this lecture was to look at some aspects which affect the long-term well-being of development projects. Once the software has been developed the emphasis moves to its maintenance. The cost of software maintenance usually exceeds the cost of software development. There are three principal types of software maintenance:

perfective - where new functionality is added to the system;

adaptive - where the system is adapted to new enviornments (e.g. ported to a new operating system);

corrective - which is removing defects from the software.

Having defined software maintenance, the lecture moved on to look at how to minimise its cost and assure successful completion of the project by asking three questions:

Why is the software process so important?

What is so good about iteravtive development anyway?

Why can't we just get on with writing the code?

To answer the first question, the most common reasons for failure of software projects were listed. An analogy based on building bridges showed how activities, such as adequate analysis and design, can be used to avoid them. The second question was addressed by giving an example of what iterative development means and by showing the unfortunate results of not using it.

Hopefully the students understood that by answering the first two questions the answer to the third becomes clear. As a means of supporting interative development cycles, configuration management systems were introduced and the lecture finished by emphasising that software always costs something (time or money): either *some* up-front by investing in analysis and design or *more* later to fix all the problems.

5. EXERCISES

The hands-on tutorials included a series of exercises to solve given problems. The tutorials followed the natural progression of physics analysis exploring the major packages of LHC++/Anaphe and ROOT on the way. The students completed the tutorials in groups of two. The students were required to develop several C++ programs in succession starting from skeletons:

1. Generate a set of events to be stored on disk according to a defined object model thereby exploring the issues of data persistency;
2. Build a set of event tags (Ntuples) from data prepared in 1 and identify/calculate interesting event attributes;
3. Use the minimization packages to find the minima values for a given set of problems
4. Read event tags built in 2 and display the contents. Use the interactive graphical tools to apply more cuts.

The LHC++/Anaphe and ROOT lectures are documented as separate papers in these proceedings. Below is a summary of the software engineering lectures by Bob Jones and the feedback session.

6. FEEDBACK SESSION

The track finished with a feedback session during which answers were given to the questions asked by the students about different aspects of the software suites covered by the track (ROOT and LHC++). The JAS software suite, though officially part of another track of the school, was included. The

students had submitted written questions to which the developers of each software suite had provided written answers. These answers were collected together and put on the school web pages. The major issues covered by the questions were:

Data storage

Interfacing to external code, experimental packages

Scaling

How does this work relate to GRID?

A subset of the questions were presented during the feedback session for further discussion. The answers provided by the authors of each software suite were shown and then follow-up questions were asked by the students. The questions were as follows:

OBJECTIVES AND APPROACH

ROOT

With the ROOT system, written in C++, we provide, among others, an efficient hierarchical object store, a C++ interpreter, advanced statistical analysis (multi dimensional histogramming, fitting and minimization algorithms) and visualization tools.

The user interacts with ROOT via a graphical user interface, the command line or batch scripts. The command and scripting language is C++ (using the interpreter) and large scripts can be compiled and dynamically linked in.

ROOT also contains a C++ to HTML documentation generation system using the interpreter's dictionaries (the reference manual on the web is generated that way) and a rich set of inter-process communication classes (supporting TCP/IP and shared memory). For the analysis of very large datasets (> TB's) we provide the Parallel ROOT Facility (PROOF).

The system is packaged in a set of modules (shared libraries) which are dynamically loaded only when needed.

The ROOT project was started in January 1995 to provide a PAW replacement in the C++/OO world. The first pre-release was in November 1995 (version 0.5) and the first public release in fall 1996.

LHC++/Anaphe

The Anaphe/LHC++ project aims at replacing the full suite of functionality formerly provided by CERNLIB. Amongst those packages the analysis tool (Lizard) is one component. It uses a set of fundamental libraries (like, e.g., HTL, CLHEP, HepODBMS) which have been developed in the last couple of years in close collaboration with experiments (mainly LHC but also other CERN and non-CERN experiments) and other HEP projects (such as Geant-4). We try to make use of good software engineering practices to improve the quality and long-term maintainability (UML, use cases, tools etc.)

The aim is to provide a flexible, interoperable, customizable set of interfaces, libraries and tools. Re-use of existing (public domain or commercial) packages as far as possible. Writing HEP specific adaptations wherever needed. Taking into consideration the huge data volume expected for LHC, the distributed computing necessary to analyse the data as well as long-term evolution and maintenance.

The use of an OO DB with transaction safety (locks) guarantees consistency in the datasets written. This is especially important in a distributed/concurrent environment. The basic

libraries exist since about 1997. Development of some parts (AIDA, Plotter, and Lizard) started in fall 1999. The first release is scheduled for October 2000.

JAS

Leverage the power of Java as much as possible because:

Provides many of the facilities we need as standard.

Is easy to learn and well matched (in terms of complexity) to physics analysis

Is a mainstream language, so time spent learning it is well spent?

Is a high performance language (see Tony's talk)

Is a highly productive language (no time wasted debugging core dumps).

JAS has been in development for 4 years (since Hepvis 96)

HOW DOES THE SOFTWARE WORK WITH NON-NATIVE DATA STORAGE?

If an experiment defines its own storage system, can the software suite use it? What capabilities are lost in that case? Specifically, can ROOT/JAS work with HepODBMS/Objectivity without losing capability? Can JAS/LHC++ work with ROOT files without losing capability?

JAS

JAS does not have a "native" data format, it can work with any data format for which a DIM exists. DIM's already exist for PAW, ROOT and Objectivity and many other formats; it is fairly easy to create new DIMs for experiment specific data.

The more detailed question is harder to answer, the specifics depend mainly on how completely the DIM has been implemented. For example the current Objectivity DIM is only able to read HEPTuple data from Objectivity databases. Objectivity does have a Java binding, so writing a more fully functioned interface is possible, although there are some complications arising when attempting to read data initially stored into Objectivity from C++, especially if no thought was given to Java access up front.

ROOT

Root can read any type of data not in Root format. The typical situation is to read ASCII files or any type of binary data via normal C++.

The h2root program is an example of a C++/Root based program converting PAW files to Root format.

NASA has implemented an interface between Root and the HDF files that are the standard for AstroPhysics.

Root has interfaces to RDBMS systems such as MySQL and Oracle. An ODBC/JDBC interface has been developed by Valery Onuchyin (see link on Root web site). We have tens of examples of collaborations using their legacy data and processing them with Root.

LHC++/Anaphe

In Lizard you have access to all the experiment's code and data in their native (storage) format using the Analyzer. If you want to store the histograms/ntuples using their own storage system, an implementation (adapter) of the Histo/Ntuple Abstract Interfaces is needed.

Questions from the audience:

Q: Can you use ROOT and Objectivity together?

A: (Rene) There are some implementations already developed.

Q: It would be useful to have a standard tool to convert ROOT files to Objectivity...

A: (Rene) We haven't receive any request for this.

Q: How is the distribution of data made in BaBar (is it Objectivity)?

A: (Bob Jacobsen) We use both Objectivity and ROOT. We have observed the same performance within 10%. We have 8 large sites using Objectivity and many small sites using ROOT.

Q: Could someone give me a definition of Objectivity?

A: (Andreas) It is a commercial product which enables the user to provide OO schemas to describe his/her databases. It describes and stores his/her data model in a completely

transparent (to the program) and location independent way. It provides all the regular database features.

LHC DATA SIZES

What will need to be developed to handle the expected size of LHC data analysis? What are the current strengths and weaknesses for storing very large amounts of data?

JAS

The strengths of JAS are in its ability to adapt to whatever data format is eventually decided upon, and to support access to very large datasets using its distributed client-server mode. There are some weaknesses in the current java.io package when dealing with large amounts of binary data, but these will be addressed by the addition of a new java.nio package in the next release of Java (JDK 1.4 scheduled for release next summer). After which there is no reason to expect Java IO will be any less efficient than C++ IO.

ROOT

Root is assumed to work in conjunction with an RDBMS. The RDBMS handles the run/file catalog and other data that require locking, journaling, etc. Root files have a current practical limitation to 2 GBytes. All the hooks are already in Root to support larger file sizes. The Alice data Challenge has demonstrated the storage of 25 TeraBytes of data with the run catalog (25000 files) stored in a simple MySQL database. A run catalog of 1000000 files has been successfully tested with MySQL. Many experiments are currently experimenting with the combination Root + RDBMS and expect to store several hundred TeraBytes in 2001.

LHC++/Anaphe

In LHC++, there were a number of studies done on how Objectivity/DB scales to store several petabytes of data. Presently, only the fixed organization of the Object-Identifier limits the amount of data that can be stored in "small" (few GB) files per Federation.

During the last year, the BaBar experiment has accumulated more than 100 TByte of data in Objectivity/DB thereby showing scaling behaviour to amounts of data that are only one order of magnitude lower than the ones expected for LHC.

As stated above, the use of an OO DataBase with transaction safety (locking) guarantees consistency in the datasets written. This is especially important in a distributed/concurrent environment.

Questions from the audience:

Q: Is it possible to store files with sizes above 2Gb?
A: (Bob Jacobsen) I don't want too many files but I don't want very big files either.
A: (Rene) ROOT has a current limitation to files of 2 GB. This is the limitation imposed in general by the OS. All the hooks are in the system to support larger files in a backward compatible way. You need different separate techniques to store run catalog and events store.
Q: How can we retrieve huge amounts of data rapidly?
A: (Bob Jacobsen) None of these software suites has addressed all the questions, but we know now what the questions are.
Q: How is Objectivity used in BaBar?

A: (Bob Jacobsen) It is used as an object store, but there is something on top to find where the event is, so that access to data is location-independent. We created an API on top of Objectivity to provide this functionality, which is needed by every experiment.

A: (Rene) A remark on performance. In ALICE simulation Objectivity and ROOT differ by a factor 5 in size. There is a factor from 3 to 5 in real time for accessing data (ROOT being the faster).

A: (Bob Jacobsen) I have a comment on benchmarks done by people who know their own system very well and learned the other one in a short time... However Babar created its own way of retrieving data.

COMPATIBILITY WITH EXTERNAL SOFTWARE

How does the software work with external software such as GEANT4 and GEANT3? What can you do and not do with them?

JAS

We demonstrated the use of JAS with Geant4 during this school. The Geant4 collaboration is considering a proposal to adopt the AIDA interface as a standard interface to histogramming in Geant4, meaning that it will be easy for Geant4 to interact with any AIDA compliant analysis tool.

We have interfaces with Root, Objectivity, PAW, WIRED, G4, StdHEP, and AIDA. Due to the simple "plugin" mechanism we expect to develop many more.

ROOT

The Alice collaboration has developed an abstract MC interface in the AliRoot framework. This abstract interface has currently an implementation (TGeant3) for Geant3 and a set of classes. TGeant3 is independent of Alice

- AliGeant4 with Alice specific classes for Geant4
- TGeant4: an experiment independent interface to G4 called by AliGeant4

The geometry input and the hits output is MC independent (same classes for Geant3 and Geant4).

A Root GUI (for GEANT4) has recently been developed in Alice by Isidro Gonzalez. This work will be presented at the coming Geant4 workshop in October.

Also in Alice, Ivana Hrivnakova who developed the AliGeant4 and TGeant4 classes is currently working in processing all the G4 classes via rootcint. This work is now close to completion. Once it is done, the following facilities will be available:

- Automatic I/O of G4 objects
- Automatic inspection and browsing of G4 data structures
- Cint interactive interface to the G4 classes.

LHC++/Anaphe

In Lizard access to Geant-4 (as to any other external software) is through the Analyzer. In addition, Geant-4 is developing AIDA based Abstract Interfaces to be used in connection with analysis packages.

As for the experiments, in the context of LHC++ we have been working closely together with Geant-4 for the design/implementation of the object persistency using an Object Database.

Questions from the audience:

Q: Can LHC++ interface GEANT3?
A: (Andreas) This can be done through the package. Be aware of limitations of using Fortran code (especially with common blocks) in multi-threaded environments which are becoming more and more popular.
A: (Rene) GEANT3 and 4 can be interfaced in the same way in ROOT. The idea to develop an abstract interface in ALIROOT was to easy the transition from GEANT3 to GEANT4. Thanks to this interface, Alice had the same geometry input and the same output objects with G3 and G4.

EXISTING EXPERIMENTAL SOFTWARE

If an experiment has an existing software package how do you interface it and how much of its capability will be available?

JAS

In principle, using a combination of plugins and DIM's you should be able to interface any experiment to JAS. In practice in depends how "Java Friendly" the experiment is (extensive use of C++ features such as templates tend to make it more difficult). Well-designed, modular experiment software also helps. The person who builds the JAS interface will need to learn a fair bit about Java and JAS, but once that is done it should be easy for other collaborators to use the interface.

Direct interface with C++ code is currently a weak point of Java, thus direct interface with C++ experiment code is currently difficult. We expect more and more experiments to adopt Java as the huge productivity benefits of using Java become more widely appreciated, meanwhile we are attempting to address this issue via the development of tools such as JACO, and plan to test this in the context of the Atlas event model. Interfacing with experiment software in Java (such as LCD) or via some intermediate storage format (e.g. PAW, Objectivity, ROOT) is comparatively straightforward.

ROOT

There are two ways to use ROOT. As a framework or as toolkit. When using it as a framework ROOT will control the event loop and call your software. This will allow you to use all ROOT's capabilities. Or you can use ROOT as a toolkit you keep your event loop and you use ROOT features like a 'sub-routine library' calling e.g. ROOT's histogramming, I/O, etc. Both approaches are used by several experiments. ALICE, STAR, etc. use the first, while ATLAS, LHCb are using the latter.

LHC++/Anaphe

If the experiment's s/w package is independent of the categories used in Lizard, its full capability is available through the Analyzer. In case the experiment wants to replace one or more of the categories, one or more Adapter to the Abstract Interface needs to be written (if not already done).

Questions from the audience:

Q: (Bob Jones) How is it possible to interface the suites with other software components (e.g. CORBA)?
A: (Andreas) We will address that problem next year.

A: (Bob Jacobsen) It has been done in Babar. We use JAS via CORBA (it is simple to interface JAVA with CORBA).

A: (Rene) There is no problem, since we already developed socket-based high performance communication. We could implement CORBA as well.

IF I WANT TO MAKE AN IMPROVEMENT, HOW DO I GO ABOUT IT?

LHC++/Anaphe

For the HEP part of Anaphe/LHC++ download the sources from the CVS repository (or the tgz file), modify and let us know!

ROOT

Proceed like several thousand people who are already doing this. Subscribe to the roottalk mailing list, see what happens and feel free to submit your complaints, additions, etc. The Root web site has a long list of links to users' contributions. The CREDITS file in the Root source distribution has a long list of people who have contributed in many different ways to the Root system.

JAS

JAS is an "Open Source" project. All of the source code is easily available and we use a Java version of make that allows you to build the system yourself, in the same way on any platform, using the simple instructions on our web site. (Building JAS from scratch takes less than one minute, and much less if you only need to recompile files you have changed). Any changes or additions you make are likely to be happily accepted back into the project.

In addition you can often extend JAS without having to learn the internals of the program by writing a plugin which adds the extra functionality you require.

The developers have made most decisions (on direction) with feedback from people using JAS for specific experiments (particularly LCD, Babar). We have a mailing list and bug report page and encourage feedback and suggestions from anyone (negative feedback is very welcome, especially if accompanied by suggestions for improvements).

Questions from the audience:

Q: What licences are needed by these software suites for HEP?

A: (Fons) In ROOT we have an Open Source approach.

A: (Bob Jacobsen, on behalf of Tony Johnson) We use an Open Source/GPL approach.

A: (Andreas) For the packages developed by us, we use an Open Source approach. For the commercial components, you have to simply register with us if you are part of the CERN research programme, otherwise you have to negociate your own licence. The exact type of licence for the packages developed by us is not yet defined, will be either GPL or something very similar.

Q: To what extent is CINT a constraint for people willing to develop new features in ROOT? If I want to write an extension to a ROOT class, should it be CINT complaint?

A: (Fons) There are no constraints as far as CINT is concerned. You only need to have code that compiles on all the platforms supported by ROOT (C++ standard compiler).

HOW DOES THE SOFTWARE UTILIZE LARGE PARALLEL FARMS FOR COMPUTATION?

JAS

JAS has been designed from the outset to run in a "client-server" mode, and to support distributed data analysis. There are Java bindings to many of the GRID components (e.g. GLOBUS) and we expect that features of the GRID such as global authentication will be easy to

interface to JAS. We believe that the model of moving the code to the data (rather than vice-versa) is most applicable to HEP data, and think Java is the best language for exploiting this due to its high performance and built-in network and code portability features. It has not yet been tested with very large datasets on large farms, but that will hopefully be done in the coming year.

ROOT

Following our long experience with parallel architectures in the early 1990s and in particular the development of the PIAF system for PAW, we developed a first prototype of PROOF in 1997 with the goal of using a parallel cluster in a heterogeneous environment. We are investing a lot of effort with PROOF to support large parallel farms. This work will be gradually integrated with the current GRID projects (e.g. Alice has submitted a GRID proposal based on PROOF).

LHC++/Anaphe

This has not yet been studied in the context of Anaphe/LHC++/Lizard. The analysis/design/implementation of this will start at the beginning of 2001 based on the Globus GRID toolkit.

CHOICE OF SCRIPTING LANGUAGE

To what extent can a user or experiment choose their scripting language (e.g. Java, Python, CINT, etc)? Can an experiment choose more than one?

JAS

First, Java is NOT a scripting language. Scripting languages are designed differently from compiled languages such as Java, C++ and Fortran, and to use a compiled language as a scripting language or vice-versa would be unwise. Having said that Java does exhibit some of the advantages sometimes associated with scripting languages, such as very fast compile, load, run cycle (especially when using dynamic loading to load only your analysis routines, as in JAS).

We are currently adding support for scripting languages to JAS; we made a demonstration of beanshell as a scripting language during the school. There are many other scripting languages available for Java, including JPython, a complete and very fast implementation of Python in Java. Any Java scripting language can be very easily used with JAS (or any Java program). There is no technical reason why an experiment should not use more than one.

ROOT

The default command/script interface in Root is based on CINT. If a user does not like CINT, he can make an interface to other languages such as Python. The Root classes may be invoked directly. It is worth mentioning that the number of requests for this solution is close to 0.

Subir Sarkar (L3 Bombay and CDF) has developed an interface to the histogramming package using the Java native interface (JNI) and also a native Java interface to the Root files.

LHC++/Anaphe

In Lizard, the scripting language can be any of those supported by SWIG (www.swig.org) including Perl, Python, Tcl/Tk, Mzscheme, Ruby and Guile. A module for Java (although this is not a scripting language) is in preparation.

QUESTION FROM TONY JOHNSON TO THE STUDENTS:

Suppose that during the CSC 2000, the authors of Root, JAS and LHC++ get together and decide they are wasting time creating similar Analysis systems. Over a late night Ouzo they all decide to work together on their next project, a system for analysing and predicting the future value of the Euro.

They all immediately quit HEP to form their new start-up company and become fabulously wealthy. You are assigned to provide support for one of these orphaned analysis packages for the next 5 years of your experiment (15 years if you are working on an LHC experiment). Which one would you choose?

Votes:

ROOT 25

JAS 17

LHC++ 4 (of which 2 were LHC++ developers)

The other students did not vote. Bob Jacobsen completed this discussion on issues covered by asking a few questions to the audience:

Are we ready for the LHC?

What's not yet understood?

What's still to be built?

SOFTWARE ENGINEERING READING LIST

GENERAL

Software Engineering; Ian Summerville; Addison-Wesley; 1995; ISBN 0-201-42765-6

A Discipline for Software Engineering; Watts S. Humphrey; Addison-Wesley; 1995; ISBN 0-7515-1493-4

Managing the software process; Watts S. Humphrey; Addison-Wesley; 1989; ISBN 0-201-18095-2

Principles of Software Engineering Management; Tom Gilb; Addison-Wesley; 1988; ISBN 0-201-19246-2

R.S. Pressman, Software Engineering: A Practitioner's Approach, McGraw Hill, 4th Edition, 1996, ISBN: 0070521824

The mythical man-month; Frederick P. Brooks, Jr.; Addison-Wesley; 1982; ISBN 0-201-00650-2

Software Quality; Joc Sanders & Eugene Curran; Addison-Wesley; 1994; ISBN 0-201-63198-9

OO ANALYSIS AND DESIGN

Principles of Object-Oriented Analysis and Design; James Martin; Prentice Hall; 1993; ISBN 0-13-720871-5

Object-Oriented Modeling and Design; James Rumbaugh et al., Prentice Hall; 1991; ISBN 0-13-629841-9

Object-Oriented Modelling with Syntropy; Steve Cook & John Daniels; 1994; ISBN 0-13-203860-9

Building Object Applications That Work; Scott W. Ambler; 1998; ISBN 0-521-64826-2

UML

Real-Time UML; Bruce Powel Douglass; 1998; ISBN 0-201-32579-9

The Unified Software Development Process; I. Jacobson, G. Booch, J. Rumbaugh; 1998; ISBN 0-201-57169-2

UML Distilled; Martin Fowler; 1997; ISBN 0-201-32563-2

Applying Uml and Patterns; Craig Larman; ISBN 0137488807

H.E. Eriksson & M. Penker, UML Toolkit, Wiley, 1998, ISBN: 0471191612

UML 2001: A Standardization Odyssey, Cris Kobryn, Comms. Of the ACM, Oct. 1999/Vol. 42, No. 10

ONLINE REFERENCES

Object Management Group http://www.omg.org

Rational Corporation http://www.rational.com

Software Development Magazine http://www.sdmagazine.com

Software Engineering Resources http://www.rspa.com/spi/

Dr. Dobbs Software Tools for the Professional Programmer http://www.ddj.com

Journal of Object Oriented programming http://www.joopmag.com/