Development of Software for ALEPH using Structured Techniques

presented by G. Kellner, CERN

Abstract:
Structured Analysis / Structured Design (de Marco, Yourdon) was chosen by the ALEPH collaboration some 2 years ago to support development of off-line and data-acquisition software in our distributed environment. After a brief introduction of the basic concepts we will show how the various methods are being applied and comment on practical experiences. A formal model is being used to describe the data and the transition from abstract data flows to real-life data will be shown. The rôle of automated tools to support the methodology will be stressed. These techniques have been accepted very positively by the physicists since they provide improved ways to communicate and document complex software issues without introducing burdensome overheads.

1. Introduction

Software systems are very often badly designed, badly or not documented, delivered too late, very costly to modify and unstable against upgrades. Our own environment, i.e. High Energy Physics, is no exception.

Why is this so? It is certainly true that concepts of software engineering were only introduced some 15 years ago. The tendency is to write code immediately - because one knows what one wants - and think afterwards how to modify it to include other options. Software writing is still an art - some people are very good at it but they usually don't (or can't) explain to other people how they do it.

Software development methodologies have been devised to bring more rigour to the various activities and provide methods, tools, and management procedures to help in producing the desired software system in time, within budgets, and with the necessary
organisation for successful operation and upgrades. There is unfortunately no unique methodology universally suitable for all kinds of projects. One has to make a careful selection of a methodology that is well suited for the project, for the people who are going to use it and for the environment of software development and operation.

We will report on practical experiences with the application of Structured Analysis / Structured Design (SASD) in a High Energy Physics environment.

2. Historical perspectives

In the 1960's it was realised that the costs of software production were rapidly escalating and had already overtaken the costs of hardware. The indirect costs were even bigger as software development was notoriously late and unreliable therefore causing serious delays in the completion of projects. Various surveys of large projects were undertaken to understand this situation. One of the results of these studies showed the fraction of time spent in different stages of the software project: analysis 30%, coding 20%, testing 50%. The fact that at the end of the coding phase only half of the project was completed explains why software always appeared 'late'. Many other problems were encountered and it became evident that a major cause of difficulty was the lack of a systematic approach to software design and development.

During this period Software Engineering was largely a research activity with little impact on current software development practices. However, some of the most important notions of modern software development concepts emerged during the late 1960's and early 1970's, namely top-down design, stepwise refinement, modularity, structured programming, and structured programming languages. Some of these concepts were applied in isolation for specific problems.

A significant new impetus came from the introduction of the 'software life-cycle' around 1975. The concepts are well known from manufacturing in which products are conceived, specified in detail, designed, built, and then maintained until they are no longer needed.

Major new ideas on 'Structured Design' were introduced in 1973 [1] and on 'Structured Analysis' in 1977 [2]. The application of the concepts of the life-cycle to software development made it possible to develop frameworks which bring together many of the known techniques with appropriate management practices in "Software Design Methodologies".

A very comprehensive collection of original papers, reviews and comparative studies can be found in [3].
3. Situation of High Energy Physics experiments

A major change has affected HEP experiments in recent years. Especially for physics with colliding beams there is a relatively small number of very large experiments with complex 4π solid angle detectors. Events are complex with typically ≥100 kbytes / event recorded and low rates of interesting events in a 'sea' of standard events. Software aspects for these experiments are also very complex: in the data-acquisition systems more and more functions and decisions are entrusted to distributed systems of micro- and mini-computers. Large volumes of data, calibration information, bookkeeping details have to be managed. The off-line data processing has to provide detailed reconstruction of trajectories and energy deposition, it has to provide detailed simulations of these interactions and sophisticated graphics programs to view and manipulate such events in order to understand what is happening. As a consequence programs require several hundred thousand lines of code and documentation.

The duration of experiments has increased to about 10 - 15 years. These very long periods imply that nothing can be assumed to be stable in any way - there will be continuous evolution in physics interest, accelerators, detector hardware, understanding of the detector responses, algorithms, programs, analysis criteria. The most important fact is that also the people who construct, operate, and maintain the hardware and software will change continuously. Many people will join an experiment at a time far beyond the initial stages of design and implementation. An essential task is therefore to make sure that these people can be made familiar with the current status of all aspects of hardware and software so that they can successfully continue the experiment. Equally important is proper documentation and reproducibility of the status at any time in the past since one will frequently have to go back to previous stages of the experiment.

One particular feature of the long preparation and duration of these experiments as well as the general economic situation in HEP is the fact that it is no longer possible to get a team of people together in one place to develop software together. Rather, it is imperative that this activity is split up into several fairly independent tasks which can then be tackled by individual smaller teams in a geographically decentralised way. Overall coordination of the total effort must ensure that the various activities fit together in a coherent way and that the project advances as planned.

4. Selection of a Software Development Methodology

In March 1983 we started to look for ways how we could best organise the development of software for the ALEPH experiment. Most of us had worked with fairly sizeable experiments in the past (though on a much smaller scale than ALEPH) and we agreed that we did not want to repeat the "standard practices" that had been used. All of us had suffered
enough from badly organised code, poor or non-existent program documentation, non-existent specifications and so on. We had been interested in this question before but we also received valuable input from the first Workshop on "Software in High Energy Physics" in October 1982 at CERN.

The technical literature abounded with articles on new methods and procedures which showed very promising results in the development of large software systems [4]. A small group of us went through this literature and collected practical facts about the bewildering number of different methodologies proposed. We were greatly helped by several comparative studies, especially on the applicability to different types of software systems, ease of learning and usage, support for various phases of the life cycle, support by automated tools and other very essential points which would otherwise have been impossible to collect in a reasonable time [5], [6].

It became evident quite soon that SASD [7] seemed to satisfy best our own selection criteria. The main reasons for this selection were: SASD covers all phases of the software life-cycle, it was widely used in industry and was therefore supported by tools originally developed for in-house use but then marketed as commercial products, it seemed better suited for our environment (the software developers are physicists not software engineers) and we felt more at ease with the basic concepts of the methodology. These views were confirmed by a Tutorial on "Software Design Techniques" given by A.I. Wasserman at CERN in June 1983.

The excellent courses given by A. Kennedy convinced the overwhelming majority of the about 50 participants that SASD could indeed be applied in our project and environment with a good chance of success. The proposition was to apply SASD to the development of software for ALEPH and this was supported by the management of the experiment towards the end of 1984.

5. Main characteristics of SASD

SASD - like other methodologies - provides a set of technical methods and management procedures supported by automated tools to cover the various phases of the software life-cycle. It should be noted that SASD has evolved considerably during the past few years due to the practical experience gained from a wide range of applications. Very recent additions include the introduction of data modelling concepts and extensions to SA for real-time applications [8].

A very comprehensive description of the methodology and its application can be found in [9]. Here we want to present some of the major concepts, show how these are applied in our experiment, and comment on our experience. Examples will be shown of the application of SASD to software development for event simulation, event reconstruction, data acquisition
and several smaller projects.

We will consider the following topics:
- software life-cycle notions
- diagramming techniques
- common data dictionary
- detailed specifications
- management procedures
- support by automated tools

6. Software life-cycle notions

The software life-cycle can be defined as the period between the initial decision to implement a software system and the end of its utilisation in operations. This period is divided into several phases, which are considered sequentially for the sake of simplicity, although they tend to overlap to a certain extent and several iterations may be required. In practice software development will be made in incremental steps. There are variations in the number and names of phases but the following main stages appear on most of them:

- User Requirements
- Analysis
- Design
- Implementation
- Testing
- Maintenance

At the begin of our software development we have produced a rather formal and concise document which stated our User Requirements and this has proven to be useful as guidance for further actions. The concept of providing a detailed analysis of the functionality of the program before major sections of the code were written certainly was the most novel approach and we found it very useful as will be shown further on. Overall and detailed Design have not been followed in all the rigour as proposed by the methodology, partly due to lack of suitable tool support. Implementation and Testing follow rather well-known patterns though many of the concepts of partitioning of work, test plan, implementation plan, and others are applied.

7. Diagramming Techniques

The power of various diagramming techniques compared to standard prose (however well structured) is demonstrated in [10]. Diagrams provide a major improvement to structure large systems into manageable portions. They allow showing of relations between different parts of a system on a global and local scale. The top-down decomposition into more and
more detail provides a very useful procedure to document the increasing understanding of the project as one advances in the development. Higher levels in a hierarchical decomposition show more abstract views of the overall project but one will find very quickly which particular route has to be followed to find the required details. At any stage the diagrams can be shown and discussed between different groups of people (team members, managers, others) since they provide an up-to-date view of the current state of understanding.

The following types of diagrams were used in our applications:
- Data Flow Diagrams (DFD), with extensions for real-time
- Entity-Relationship Diagrams (ERD)
- Structure Charts (SC)

7.1 Data Flow Diagrams

The DFDs provide a graphical representation of a model of the system to be produced. They show the flow of information through the system, the processes which transform input data flows to output data flows, the data stores as a repository of data, and the external sources and sinks of data. The DFD allow the functional decomposition of the complete system and all the interfaces between the processes to be shown in the form of a network. Large systems will contain many data flows and processes. Each process can in turn be decomposed into a DFD at a lower level (fig.1) until a level is reached where processes can be described in textual form as Process Descriptions.

The following figures show several examples from our applications. Fig. 2 shows the overall view of the ALEPH software: data acquisition, event reconstruction, physics analysis (including interactive graphics) and event simulation. The following features are noted at a glance: a) all programs make use of a common data base to extract constants and store bookkeeping information, and b) programs are connected since output from one program can be used as input to other programs. Figs. 3, 4 and 5 show the decomposition of process 4 of fig. 2 to lower levels. Fig. 5 actually represents the lowest level of decomposition of this particular branch.

One of the basic concepts of Structured Analysis requires that only data flows and processes are shown on DFDs and that all control information shall be eliminated at this stage. This looks quite reasonable for programs which are basically batch oriented. However, it causes problems for the design of real-time systems (e.g. data acquisition) or highly interactive systems (e.g. interactive graphics). Since 1982 new concepts were gradually introduced which meet the same goals as the basic SA, but put equal emphasis on process and control [8].

Fig. 5 shows as an example a part of the system that provides operator control over data-taking activities. Control flows and control processes on the DFD are indicated with
dotted lines to distinguish them from 'normal' data flows and processes. The control logic can be extremely complicated and errors can easily be introduced. State Transition Diagrams or State Transition Tables provide ideal techniques to show all possible states. The fourth diagram (Structure Chart) shows the actual implementation. An overview of these extensions and their practical impact on the design of the data acquisition software for ALEPH will be presented by T. Charity [11].

7.2 Entity-Relationship Diagrams

ERDs provide a convenient way to graphically depict data structures and specific properties by showing the entities and the relations between them. ERD were originally not part of the standard methods provided by SA. The development of ADAMO [12] introduced the Entity-relationship model and it has since been adopted as the standard way to describe all data in ALEPH. Since all information about data is contained in the data dictionary ERD can be used to display any conveniently selected subset in graphical form. The example on fig. 6 shows a partial ERD for the ALEPH detector description restricted to the ITC (Inner Tracking Chamber) and TPC (Time Projection Chamber).

7.3 Structure Charts

Structure Charts (SC) provide a model of an implementation of a system in terms of software modules (functions and subroutines in FORTRAN). Data flows and control flows between modules are shown and extra procedural information can be indicated. Objective criteria allow evaluation of the design. Specific questions of repackaging, access procedures and information hiding modules (e.g. for complicated access to data base information), compromises on otherwise good designs for efficiency reasons, real-time and hardware constraints can all be shown explicitly on the SC with a few additional symbols.

Our present experience with SC is limited. In certain cases partial SC have been drawn and they have indeed been very useful to find more appropriate calling sequences or group together a set of modules in an access package. Further work will be needed to decide in which way SC will be employed. An example of a (partial) SC is shown in fig. 7. This represents the top of the reconstruction program with the main input and output modules. This structure serves as a common 'skeleton' for the implementation of further modules.

8. Common Data Dictionary

The data dictionary, as it is used in Structured Analysis, records the information content of data. It contains definitions of all data mentioned in the DFD, in process specifications, or in the data dictionary itself. Composite data flows are used on the higher levels of DFD to reduce the number of flows and they can be defined in terms of their components. At some stage in the decomposition data flows can no longer be divided into other flows but rather
into a set of data elements. Later in Analysis, or in Design, further secondary information about data may be added (e.g. validity ranges, access methods, frequency). The standard syntax used in many text books and in certain tool packages did not appeal to us, however.

It turned out to be rather straight-forward to extend the syntax used by ADAMO to include data flows as well. Flows which can not be decomposed any further correspond to the entity sets of ADAMO and the data elements are the attributes of that entity set. The logical data structures can be described by indicating relationships between entity sets. Our particular set of conventions allows long names in "ZurichStyleNotation" for data flows (up to 31 characters) and up to 16 characters for entity set names, attribute names and relationship names. An example of the DDL syntax is shown in fig. 8.

It should be stressed again how important it is to keep all data in a project defined together in a computer-readable data dictionary as unique source of the information. Version control of the data is then almost trivial. The same definitions of data will be used in all programs and only one source has to be updated. Utilities can be written to display the data in various ways.

We have decided to store all our data into a common data dictionary expressed with the Entity-Relationship model and described in ADAMO syntax. This includes data flows from the DFD, entity and relationship sets as used in the Fortran code, detector descriptions and other database information.

9. Detailed Specifications

The lowest levels of a set of DFDs or a Structure Chart partition a system such that each part can be specified concisely and independently of the other parts. These specifications usually require about half a page to state the policy.

The most successful method for descriptions is structured English, also called pseudocode or PDL (Process Description Language). This is a restricted natural language with additional keywords to provide structuring (constructs familiar from structured programming are used). Some policy decisions are not easy to express in structured English alone. For specific cases where many logical conditions are possible (e.g. combinations of independent variables) it is better to show these with decision trees or decision tables. State Transition Diagrams or State Condition Tables are used for interactive transactions.

We have used Process Descriptions in PDL for a number of applications. The general tendency was rather to skip the PDL and write useful comments directly into the Fortran subroutines, together with standard header information which could be extracted as description. Expansion of the PDL by inserting the actual code inbetween lines of PDL was found to be quite unreadable.
10. Management procedures

Several procedures are provided to verify that the 'product is developed right' and to validate that the 'right product is produced'. This includes a set of criteria to check the completeness and correctness of the various DFD, DDL and PDL produced.

The 'walkthrough' provides a very powerful method for continuous verification of the quality of the product being developed. The author presents a diagram (data flow diagram or structure chart), description (process description, module description, other text), or code to a small group of people for review. He explains this by 'walking through' the diagram or text. The reviewers pose questions or point out obscure or inconsistent use of agreed procedures and standards. Errors will be corrected by the author and a new 'walkthrough' will be organised, with a further iterations if needed.

This proves to be indeed very useful. How is it done practically? One week every 4 weeks most members of the development team of the reconstruction programs will gather. Small subgroups are formed to review diagrams, data descriptions, code, documentation, and various conventions. A summary of observations is presented at common meetings. The major part of the available time is devoted to such reviews, update of information, integration into the overall skeleton, and tests on various computers (IBM/VM and Vax/VMS at present). A new release of documentation and code is usually available after this week.

A series of procedures is proposed for the setup of a 'test plan'. All the major elements of the system, i.e. a logical top-down decomposition of the modules and detailed definitions for input and output data are known at the time of the Design phase. Therefore a set of standard test cases can be set up for suitable configurations of modules and test data sets can be prepared before any code is actually available. Often this is done by an independent team.

We have not yet started to set up the test plan for a large project like the reconstruction program. However, for the development of the ADAMO code such a test plan has been defined and DTM (Dec Test Manager) is being used. More investigations are needed.

11. Support by automated tools

The availability of a set of automatic tools had been a prerequisite. It had been evident from the initial tests that tools are essential in order to help people using the methods effectively. It is possible to draw a few diagrams with paper and pencil. However, it gets very boring and people tire very quickly of rubbing out sections of diagrams or redrawing them from scratch. Tools have to provide graphics editors which allow creating and modifying diagrams quickly and easily and allow for rapid navigation between levels of diagrams and related information.
Equally essential are formal verification for correctness and completeness of the information on diagrams, specifications, and data descriptions and their cross-relations. Manual checking of this information is very tedious and error-prone and it simply consumes too much time which should rather be used to create good designs. Integration of all tools via a common database is essential to go back and forth between information relating to various phases of the project. Another important aspect is an 'open architecture' of such packages in order to allow users to integrate their own tools into the overall framework.

In the past 2-3 years there have been dramatic changes due to the development of reasonably priced workstation hardware. Development of databases and sophisticated user interfaces which exploit features of the new hardware allow creation of well-integrated sets of software development tools which are just appearing on the market. This may finally provide the real breakthrough for a wider application of software development methodologies. Several companies are experimenting with the use of expert systems to aid in the development of software systems [13]. The ultimate goal is to generate code automatically from a well structured design. It is likely that some considerable time will elapse before such products may come into wide use.

A word of caution may be useful: the methods and tools allow to make beautiful designs and layouts but there is no way that they can check if all that makes really sense - this is still the domain of human beings, fortunately.

Rather few packages were available commercially for the support of the full SASD methodology in 1984. Several more supported the Analysis phase and extensions for the Design phase were promised. We finally selected to use a commercial product from Tektronix [14]. It was available on Vax and could be used with a variety of terminals. It provided a fast and easy to use graphics editor. The source of diagrams as well as DDL and PDL were stored in standard ASCII files which could therefore be accessed and modified via standard networks. This was important for teams working at different locations. The support provided for data descriptions was not very convenient, however.

We therefore decided to extend the syntax of the DDL used for ADAMO to include dataflows and provide tools to integrate DFD's created via the Tektronix tools and DDL. The connections between ADAMO tools and Tektronix SA Tools are shown in fig.9. CSP is a program that checks and converts DDL to an internal format (the ADAMO Data Dictionary, ADD) which is more suitable for manipulation by programs. LID lists in tabular form the contents of the data dictionary ADD. TEA is a simple conversion tool which converts the output file from the Tektronix SA Tools graphics editor. DFA checks for internal consistency of the data flows in the hierarchical DFD decomposition and their compatibility with information in the ADD. The description of the diagrams is added to the ADD. DFP plots DFDs interactively on terminals with GKS drivers or in batch mode. The GKS metafile can
be used for plotted output. Some very simple tools have been written to aid in the preparation of the DDL.

12. Practical experience - Highlights and Problems

In this section we would like to sum up our practical experience in using SASD for about two years for several projects.

Positive aspects of using the methodology:
- there is a common style of work adopted and understood by everybody which makes it easier to obtain a more coherent product despite the many people with their individual 'tastes' who contribute to it

- the idea to spend a sizeable fraction of the software development in analysis of the problem is really a very positive aspect because it forces people to get clear ideas what they intend to do and document it so that other people can discuss it

- documentation via various diagrams and descriptions provide powerful means to abstract and show details wherever required

- the fact that documentation is available even before code is very useful indeed

- frequent walkthroughs and reviews provide verification and validation all along the development process, in principle one should have better programs at the time of startup of production

- the partitioning of work for several teams working independently is useful in our current situation

- the definition of all data in one unique source and in a standard way should bring enormous benefits in the long run since this is one of the more tedious problems from the past

- the method is flexible enough to be adapted to specific needs and it is reasonably easy to learn and use

Inconveniences or problems encountered:
- one of the most vexing problems was the non-availability of adequate tools - either because they do not exist yet or are too expensive or exist on certain computers or PC only, hopefully this will change in the very near future

- one has to invest money and manpower to teach courses, learn the methods well and
make sure that a consistent standard of quality is kept by investing more manpower and time for the review sessions

• while the methods and procedures used for Analysis are quite powerful and easy to use, especially with the recent extensions for real-time application, the methods used during the Design seem to be much more tedious and labour-intensive since they can not so easily be supported via tools

• the methods are flexible and can be adapted - but this sometimes creates problems because there are different ways to express things and one may waste time by fiddling around

13. Acknowledgements

Introduction of new methods and change of work style need the commitment of many people otherwise this would simply not be possible. I would therefore like to thank all people who went to the pains to learn about the new methods and employ them successfully. However, I would also like to thank those people explicitly who contributed a great deal of enthusiasm and constructive ideas to adapt the methods to our environment and projects, namely P. Palazzi, S.M. Fisher, R. Brazioli, W. Zhao, J. Knobloch, M. Green, J. Bunn, R. Fantechi, J. Harvey, R. McClatchey, and in particular also F. Dydak who gave us very strong support.

References

Summary", idem, pp.51-97


E. Yourdon, L.L. Constantine, "Structured Design", Prentice-Hall, 1979


[10] a useful survey of various techniques can be found in : J. Martin, C. McClure, "Diagramming Techniques for Analysts and Programmers", Prentice-Hall, 1985


[12] P. Palazzi, "Use of the ADAMO data management system within ALEPH", presentation at this Conference


some of the tools available for support of SASD are :
Analyst/Designer Toolkit, available from Yourdon
ARGUS II, developed by Boeing Computer Services for internal use
Excelerator, available from Index Technology Corporation
ProMod, available from GEI
SA Tools and SD Tools, available from Tektronix
PCSA, available from StructSoft
Teamwork/SA and /SD, available from CADRE Technologies Inc.
**Figure Captions**

fig. 1  Notation for System Decomposition into several levels of DFD
fig. 2  DFD showing a global overview of the ALEPH software development
fig. 3  DFD showing a global view of the Event Simulation program
fig. 4  DFD showing details of simulating analog signals in electromagnetic calorimeter
fig. 5  Overview of extensions to SA for real-time applications, showing a Data Flow / Control Flow Diagram, Structure Chard, State Transition Diagram and State Transition Table
fig. 6  Entity-Relationship Diagram showing a partial description of the ALEPH detector
fig. 7  (Partial) Structure Chart showing the 'skeleton' of the Reconstruction program
fig. 8  Syntax of DDL (Data Description Language) using ADAMO conventions
fig. 9  DFD showing connectivity between ADAMO tools and ε commercial package
fig. 1  Notation for System Decomposition into several levels of DFD
fig. 2 DFD showing a global overview of the ALEPH software development
fig. 3  DFD showing a global view of the Event Simulation program
fig. 4  DFD showing details of simulating analog signals in electromagnetic calorimeter
fig. 5  Overview of extensions to SA for real-time applications
fig. 6  Entity-Relationship Diagram showing a partial description of the ALEPH detector
fig. 7  (Partial) Structure Chart showing the 'skeleton' of the Reconstruction program
DEFINE DATAFLOW
SimulatedEvent
: 'results of Monte Carlo pass, only part of the program may be run and/or results from
previous passes may have been used as input'
= [McRunHeader], {McRunSummary}, McPrimaryEvent, McTrackedEvent,
McAnalogEvent, McDigitisedEvent, McTriggerEvent;

McPrimaryEvent
: 'all particles created at the simulated primary interaction'
= [McEventHeader], {MVER}, {MTRA}, {MVER -> MTRA};

McTrackedEvent
: 'all particles created at the simulated primary interaction and secondary interactions/decays
along the particle trajectories through the apparatus'
= [McEventHeader], {MVER}, {MTRA}, {MVER -> MTRA}, {MTEL},
{MTRA -> MTEL};

McVertex
: 'this is an entity set, but since it is also used directly as a dataflow on DFD it is also given
with a long name, similar for McTrack, McTrackElement, etc.'
= {MVER};

McTrack={MTRA};

McTrackExtrapolated
: 'example for a selector, i.e. McTrack which have been extrapolated'
= {MTRA | Extrapolated};

McTrackElement={MTEL};

DEFINE ESET
MVER
: 'all primary and secondary vertices created by Monte Carlo'
= (X, Y, Z, Vtype, Id, Chi2, Ntrack, Error(9), Code);

MTRA
: 'parameters of created tracks'
= (Px, Py, Pz, Energy, Charge, Id, Ttype, Chi2, Error(12), Label);

DEFINE ATTRIBUTE
X = REAL [-10.,10.], 'coordinate x';
Y = REAL [-2.5,2.5];
Id = INTE, 'identifiers';
Energy = REAL [0..2000.], 'Energy in GeV';
Ttype = INTE 1 | 2 | 3 | 11 | 12, 'track type, 1=TP+IT+MX, 2=etc.';
Vtype = INTE 1 | 8, 'vertex type, 1=primary, 8=V0';
Label = CHA4, 'label for printing';
Code = BITP, 'vertex fit codes, see xxx';

DEFINE RSET
(MTRA [0,1] -> [1,*] MVER)
: 'any comment, if wanted, this rset has chisquare as an attribute'
= (Chi2);

(MTEL [0,*] -> [1,*] MTRA);

fig. 8 Syntax of DDL (Data Description Language) using ADAMO conventions
fig. 9  DFD showing connectivity between ADAMO tools and a commercial package