ABSTRACT

Many studies by academic institutions, government organisations, and industry, have attempted to define improved methods to control the software development process over the past 20 years. Following a brief historic overview, the concepts of software life-cycle, modelling of functional descriptions, modelling of data, diagramming techniques, amongst others, will be described in some detail. Software Development Methodologies - which combine various methods, management procedures, and tools - have been developed in recent years and are now widely used in industry. A short review of the major methodologies will outline highlights and problems. Research is actively pursued to improve the current methods and several of the major research areas will be reviewed.

1. INTRODUCTION

In this paper a brief overview will be given of various basic concepts applied in modern techniques of software development. There exists no unique methodology which could be applied equally well to development of software for engineering, scientific, or business applications. Existing methods are being adapted and new concepts are being introduced to improve the process of software creation and maintenance. Many of the key concepts appear in most of the existing methodologies, however.

Examples shown will mainly refer to one methodology, Structured Analysis / Structured Design (or SA-SD), which is fairly widely used in the engineering environment [1]. However, no attempt will be made to provide a coherent overview of SA-SD. Examples can be found in contributions by J. Harvey [2] and S.M. Fisher [3].

2. WHAT IS SOFTWARE ENGINEERING?

The role of basic sciences - mathematics, physics, chemistry, geology, and others - is to uncover basic principles. Existing knowledge is gathered and interpreted in models to provide new predictions. Experiments are performed to prove or disprove these hypotheses, eventually leading to refinements in existing models or to entirely new insights.

Engineering disciplines are founded on one or several of the basic sciences. Well founded principles are applied to design and implement specific applications (e.g. bridges, space
probes, superconducting magnets, drugs, home appliances). The intention is certainly not to uncover new principles or disprove a prediction. Rather one is interested to create products within prescribed standards of quality, time and cost.

Software Engineering is based on general principles of mathematics and computer science. It is a comparatively young discipline compared to Mechanical or Civil Engineering. Computers started to get widely used in the early 1960’s. Many of the most important notions of modern software development concepts were being developed in this period, namely top-down design, stepwise refinement, modularity, structured programming, new programming languages, data modelling. Few, if any, of these concepts were applied in isolation for specific problems during this period.

It was realised very soon that costs for software production were escalating rapidly and that software was notoriously late and unreliable. It became evident that a major cause of difficulty was the lack of a systematic approach to software design and development.

Major new ideas on Structured Design [4] and on Structured Analysis [5] were introduced in 1973 and 1977, respectively. A significant new impetus came from the introduction of the Software Life-cycle concepts in 1975. These concepts are well known from manufacturing in which products are conceived, specified in detail, designed, built, and then maintained until they are no longer supported. A very comprehensive collection of original papers, reviews and comparative studies can be found in [6]. The application of the concepts of the product life-cycle to software development made it possible to develop frameworks which bring together many of the known techniques with appropriate management practicies in Software Development Methodologies.

The goals of Software Engineering are: to introduce an "engineering-like" discipline to improve the process of software production, to improve the quality of software products, to produce software which satisfies precise needs and to manage the complexity of large software systems. Improving software productivity is imperative to satisfy the ever increasing demand for software and to manage the increasing cost of software development as has been demonstrated by B.W. Boehm [7].

Fig. 1 Software cost trend

Fig. 2 Growth in software demand
Figure 1 shows recent and predicted software cost trends in the USA and worldwide, while Fig. 2 shows the growth in software demand for several US spaceflight programs as an example.

A very extensive discussion on Software Engineering and various approaches to solve the problems can be found in [8].

3. SOFTWARE LIFE-CYCLE

The period between the initial decision to implement a software system and its final end of utilisation is called the software life-cycle. The concept allows to structure the process of software creation into several distinct phases. The number of phases and their names may vary with different models but boundaries between different phases are clearly defined. The major deliverable products (documents, code, test results, reports, etc.) are identified and provide an essential means to measure actually achieved goals against predictions. A typical life-cycle model may include the following phases:

User Requirements or Feasibility Study phase. Some people consider this phase outside of the life-cycle. However, it is a necessary preliminary step in order to define the needs of the users of the software system. It is essential to get a good understanding of the problem to be solved. Generally this is a rather informal document.

Software Requirements or Analysis phase. The aim is to produce a functional specification of the requirements of the system. It includes the user requirements, external requirements, and constraints (e.g. external interfaces, performance, environment, safety). Requirements analysis provides the software designer with a representation of information and function that can be translated into data, architectural, and procedural design. This creates a logical model to show what the system should do and defines the information domain that will be treated by software. As the size of the problem grows, the complexity of the analysis task also grows. Therefore the system must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion. In many instances it may be difficult to specify requirements in detail, especially if a system is to be used by a wide spectrum of users with different (often conflicting) requirements. In such cases a model of the software (or a subsystem), called a prototype, is constructed to assess alternatives. This should finally result in formalized requirements.

Design phase. The aim is to translate the functional specification of the system derived during the Analysis phase into a solution in terms of software modules, i.e. how the system should be implemented. From the project management point of view software design is conducted in 2 steps. Overall design (also called Architectural design) is concerned with the transformation of requirements into data and software architecture. This creates a hierarchy of components with their interfaces. Detailed design focuses on refinements to the architectural design that lead to detailed descriptions of data and data structures and algorithmic representations in software modules. The quality of the decomposition is evaluated against a set
of criteria (e.g. coupling and cohesion of modules). Optimisation and repackaging are used to satisfy certain constraints (e.g. timing constraints, external packages or libraries, interface routines).

**Implementation and Testing** phase. The aim is to translate representations of software derived in the previous steps into a form that can be "understood" by the computer. This includes coding of modules, using principles of structured programming and coding standards. Testing includes testing of individual modules, testing of groups of modules in incremental and phased approaches, and acceptance testing of overall system.

**Operation and Evolution** phase. Often his is also called the Maintenance phase. Its main purpose is to cater for operation of the software system in the production environment. Potentially this can be a very long period and past experience has shown that over 60% of all software cost may be spent during this phase. The basic reason is that computer programs are always changing. There are bugs to fix, enhancements to add and optimizations to make. The environment or the application may also change requiring upgrades to software. These upgrades may require changes to code and documentation, changes to design issues, or even redefinition of requirements for the system. Various phases of the software life-cycle will be initiated to re-engineer the subset which requires updates.

4. **MODELS FOR SOFTWARE DEVELOPMENT PROCESS**

The primary purpose of models for the software development process is to define the order in which the various phases of the software life-cycle should be involved in the software development and maintenance. They allow to establish transition criteria for progressing from one stage to the next. These include criteria for alternative options and entrance criteria for the next stage [9].

A very simple model is being used since the earliest days of software development: write some code, fix the problems in the code. It is quite evident that there are many difficulties associated with this very simple approach. After a number of fixes the code becomes so poorly structured that even the author may have problems to know what is happening. Since the code is probably not too well structured to start with it gets increasingly expensive to maintain it and test modifications. Even before the introduction of the software life-cycle several attempts had been made to define an improved way to develop large software systems. The Software Requirements and Design phases introduced by the software life-cycle provided the necessary means for a better organisation of the development process.

Another software process model which is being widely used is the "waterfall model", shown in Fig. 3. It includes all phases known from the software life cycle. Each stage includes a step for verification ('ensure that the software correctly implements a specific function') or validation ('ensure that the software built corresponds to the user requirements'). The feedback loops between stages ensure that costs for rework are minimised. It had been shown that the relative cost to fix errors introduced in early phases of a project increase dramatically the later they are discovered in the project [9]. This is shown in Fig. 4.
Fig. 3 The waterfall model of the software life cycle

Fig. 4 Increase in cost to fix or change software throughout life cycle
The waterfall model has encountered a number of problems and criticisms. The main objection is that the model is too 'strict' - it requires that a stage is completely finished (with full documents and formal review) before the next stage can be started. In many larger projects it has proven quite difficult to accomplish this. Various extensions have been proposed to accommodate intermediate steps to include prototyping, evaluation or selection of alternatives, incremental and parallel development of subsets.

The "Spiral Model" of the software development process has evolved over the past few years, based on experience with various refinements of the waterfall and other models [10]. The model can equally well be applied to development and maintenance and it can accommodate most other models as special cases. Fig. 5 shows a rendering of the model as discussed in detail by B.W. Boehm in [10].

The main underlying concept is that there are several stages where objectives, alternatives and constraints are determined, associated risks are evaluated via prototyping, simulation, benchmarking, and other techniques, and resolved. Depending on the remaining risk factors a new cycle of the spiral may be started to evaluate and resolve more of these risks. Eventually the basic waterfall model with requirements analysis, overall and detailed design, etc. will be followed. Depending on the complexity of the project and the risks involved only a subset of all the potential steps may actually be implemented.

![Spiral model of the software process](image)

Fig. 5 Spiral model of the software process
5. **DIAGRAMMING TECHNIQUES**

"A picture is better than a thousand words". Though coined in a very different context this statement is equally well applicable to software development. Complex activities and procedures are easier to describe in graphical form than in standard prose (however well it may be organised). Diagrams represent an essential communications tool which enables developers to interchange ideas and uncover poor or wrong structuring early in the development process. At any stage diagrams can be discussed between different people (members of the team, managers, customers, hardware people) since they provide an up-to-date view of the current state of understanding. They represent an essential documentation aid in the maintenance of programs and for upgrades. Diagrams are well suited for interactive manipulation on computer graphics screens. This speeds up the process of creation and modification, it enforces standards through validity checks, cross references and calculations performed by the computer. It alleviates a lot the rather boring job of manual verification.

Many different diagramming techniques have been developed to describe process structures, control structures, and data structures. A detailed discussion of various diagrams has been provided by J. Martin and C. McClure [11]. Figure 6 shows examples of different diagramming techniques for description of data and activities.

![Diagram showing different diagramming techniques and their applications](attachment:diagram.png)

*Fig. 6 Areas in which different diagramming techniques are applicable*
Figure 7 shows a summary of the capabilities of diagramming techniques to describe certain functions.

<table>
<thead>
<tr>
<th>WHAT CAN BE DRAWN WITH THE TECHNIQUE?</th>
<th>DECOMPOSITION DIAGRAMS</th>
<th>DEPENDENCY DIAGRAMS</th>
<th>DATA FLOW DIAGRAMS</th>
<th>ENTITY-RELATIONSHIP DIAGRAMS</th>
<th>DATA STRUCTURE DIAGRAMS</th>
<th>DATA NAVIGATION DIAGRAMS</th>
<th>IPO CHARTS</th>
<th>STRATEGY CHARTS</th>
<th>WERNER-JOHNSON CHARTS</th>
<th>FLOWCHARTS</th>
<th>STRUCTURED ENGLISH</th>
<th>NASSI-SHNEIDERMAN CHARTS</th>
<th>ACTION DIAGRAMS</th>
<th>DATA BASE ACTION DIAGRAMS</th>
<th>DECISION TREES AND TABLES</th>
<th>STATE-TRANSITION DIAGRAMS</th>
<th>HOS CHARTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enterprise model showing corporate functions</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional decomposition species I (tree structure only)</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional decomposition species II (tree structure plus input and output)</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional decomposition species III (axiomatic control of decomposition)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interaction between business events</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flow of data</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonprocedural (compound) data-base actions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control structures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditions</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case structure</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Repetition</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop control</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Good for showing complex logic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Designed for showing highly complex decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linkage to data model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linkage to fourth-generation languages</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree structured data</td>
<td></td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plex structured data</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derived data items</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corporate data models</td>
<td></td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data-base navigation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7 Summary of the capabilities of diagramming techniques

Although many more diagramming techniques are currently used we will show examples of a small subset only. We will restrict to engineering environments only.

5.1 Data Flow Diagrams

Data Flow Diagrams are used during the Analysis phase of SA-SD [12]. 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.

Figure 8 shows an examples from our applications. It 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 (partial) input to other programs. This provides consistency between different parts of the software, and eases the problems of maintenance of a large number of parameters. Common data structures are defined throughout the system to simplify communication and transfer of data between processes.

Fig. 8 Data Flow Diagram showing principle components of ALEPH software

5.2 Extensions for real-time Analysis

One of the basic concepts of Structured Analysis required that only data flows and processes are shown on DFDs and that all control information was eliminated at this stage. This looks quite reasonable for programs which are basically batch oriented, e.g. Monte Carlo Simulation or Event Reconstruction. 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 who meet the same goals as the basic SA, but
put equal emphasis on process and control [13].

In a data acquisition environment the state of the whole system can change due to the detection of a particular event in the system's environment and this can be made to affect the way the system subsequently behaves. It is extremely important for the technical methods to handle these features correctly as they have a major influence on the behaviour of the system. This situation typically occurs in interactive applications where the operator can change the state of the system by the press of a button. Figure 9 shows an example of a CFD (Control Flow Diagram) describing that part of a system for providing operator control over datataking activities. Following the notations of Ward and Mellor [13] events are represented by dotted lines to distinguish from the data flows represented by solid lines and the control transformation ("Invoke DAQ Request") is represented by a dotted circle. The control transformation appears as a transaction centre which must keep track of which state the system is in and use this information together with the input control flow to determine which action procedure to activate.

Fig. 9 Control Flow Diagram of an interactive application (Ward-Mellor style)
The control logic for implementing the transaction centre can be extremely complicated and experience has shown that it is in this area that errors are introduced which are most difficult to detect in the code. State Transition Diagrams (STD) provide an ideal technique for specifying this control logic as it provides a method for representing the states of the system, the conditions for transitions between states and the actions to be performed on making the transitions. Figure 10 shows the STD for the example shown in Fig. 9.

![State Transition Diagram](image)

Fig. 10 State Transition Diagram for specifying transaction centre ('Invoke DAQ Request')

5.3 SADT diagrams

SADT, Softech's Structured Analysis and Design Technique, has been published in 1977 [5]. Since then the methodology has been applied to a wide variety of major projects in a broad range of application areas [14]. The graphical representation is shown in an example in Fig. 11 [15] which describes a model for software development. The diagram shows more details of process "development" from a higher level diagram. As in the case of DFD's this diagramming technique allows a hierarchical top-down decomposition of the whole system into easy to grasp pieces. In the activity diagram the boxes represent 'actions' or 'processes' which transform input and control to output. The arrows represent interfaces between the boxes for 'inputs' (enter box at left), 'output' (leave box at right), and 'control information' (enter from top).
Outputs from one box can get inputs or control information for other boxes. Arrows entering the box from below represent 'mechanisms', i.e. connections from other boxes anywhere in the system or external to the system. Note, that there are no data stores used in this technique. Another form of the same diagram, the data diagram, uses an orthogonal view - boxes describe the states of things and the arrows represent the activities. Both types of diagrams are required to describe a system completely.

Fig. 11 SADT diagram (at first level of decomposition)

5.4 Entity-Relationship Diagrams

Entity-Relationship Diagrams (ERD) provide a comprehensive way to graphically depict data structures and specific properties by showing the entities and the relations between them. ERD have been included recently in SA. Since all information about data is contained in the data dictionary ERDs can be used to display any conveniently selected subset in graphical form.

Two of the LEP experiments (ALEPH and DELPHI) each have over hundred crates of FASTBUS readout electronics attached to their detectors. Figure 12 shows an extract of the FASTBUS data model describing the physical layout and addressability of FASTBUS modules.
Fig. 12 Extract from the Entity-Relationship diagram used to model Fastbus

The data used in the online system can be subdivided into several schemata. These include descriptions of the readout components (FASTBUS), the detector control components (Slow Control), the data taking environment (Run and Partition) and the overall Detector description. The example on Fig. 13 shows a partial Entity-Relationship Diagram for the ALEPH detector description with only two of the nine components shown explicitly.
5.5 Structure Charts

Structure Charts (SC) provide a model of an implementation of a system in terms of software modules (e.g. 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 database information), compromises of good design for efficiency reasons, real-time and hardware constraints can all be shown explicitly on the SC with a few additional symbols. An example of a (partial) SC is given in Fig. 14 which shows the program structure derived from the CFD of Fig. 9.
6. **FUNCTIONAL DECOMPOSITION**

Complex systems can not be described in a single phrase or a single diagram. Rather, some overall decomposition of the system into smaller pieces has to be provided.

Usually the first step in the decomposition of the system is to delineate the context of its environment, i.e. everything with which the system must interact but which is not part of the system itself. This overall view will be refined in successive steps of decomposition which show more and more details of the functions and the input data which are transformed by these functions into output data, possible under some control. This decomposition is continued until the individual processes represent 'functional primitives', i.e. simple procedures, mathematical algorithms, etc. The result is a hierarchical, top-down decomposition of the whole into easy to grasp pieces which generally perform a single function.

Diagramming techniques are very well suited to structure large systems into manageable portions. They provide a 'blue-print' of the system to be built. Figure 15 shows schematically the decomposition process for data Flow Diagrams. Each process on a higher level may be decomposed further to show more details. The basic reason for this procedure is that diagrams have to be simple to read in order to be useful at all. A total of $5 \pm 2$ different objects on a single diagram seems to be a practical limit.

![Diagram](image_url)

**Fig. 15 Notation for Data Flow Diagram decomposition into several levels of DFD**

Figures 16 and 17 show examples of decomposition for SADT diagrams. Figure 16 represents the top level diagram describing the software environment for software produced for ESA [15]. The expansion of the process 'Development' has been shown in Fig. 11.
Fig. 16 SADT diagram for top level of project

Figure 17 represents a further level of decomposition of the first box of Fig. 11.

Fig. 17: SADT diagram for decomposition into next level
Examples for functional decomposition used by other diagramming techniques can be found in [11].

7. **FUNCTIONAL SPECIFICATION**

The lowest levels of a hierarchical decomposition represent 'functional primitives', i.e. single functions, in general. They can be specified concisely and independently of the other parts since all the data interfaces are defined. Various approaches to functional specification exist:

*Natural Language Specifications* use a semantics based on informal language. A widely used method is PDL (Program Design Language), also called pseudocode or Structured English. In general this is a restricted natural language (English mainly) with aditional keywords to provide structuring (constructs familiar from structured programming are used). There are many ways to define pseudocode. The more relaxed form is easy to write but, being less precise, is less useful and more difficult to process by automated tools. In principle, such specifications are easier to understand for the non-expert but they are inherently ambiguous. All checking and tracing with requirements has mainly to be done 'by hand'.

*Mathematical Specifications* use a semantics based on a proof system. Proofs are used to discover inconsistencies and to derive consequences of the specification. These are helpful for validation and checking of completeness. A discussion on formal methods for program specification has been presented by C.A.R. Hoare [16].

*Operational Specifications* use a semantics defined in terms of an execution model. They are analyzed, checked for consistency, and validated by static analysis based on the execution model or by execution itself. Descriptions using State Transition Diagrams (see Fig. 10) or Petri Nets provide examples of this.

In practical applications for different methodologies these approaches are combined in various ways. In SA-SD some aspects are represented in an operational specification (e.g. State Transition Diagrams, Process Activation Tables) [17], others use informal language (e.g. process and module specifications in PDL). In VDM [18, 19] important components of a specification are both in English and in mathematical equations. Parnas [20] uses a combination of English, State Transition Diagrams and equations to specify the A7 project. Specifications in SREM [21] and PAISley [22] are primarily operational, but mathematical proofs are used for certain validation aspects.

A very comprehensive comparison of various methods used for Software Requirements Specification can be found in [23].

8. **DATA MODELLING AND COMMON DATA DICTIONARY**

Data-flow oriented analysis methods (SA-SD) provide functional decomposition of a system into lower level diagrams which show more and more details of the functions and their interfaces (see chap. 6). As the functions are decomposed also their associated data flows may get decomposed. The higher level diagrams show rather abstract 'composite' data flows. The
lowest levels show generally data flows which can no longer be decomposed into other dataflows.

It had been realised for quite some time that the gradual decomposition top-down did not quite correspond to a realistic situation. In general, much more was known about data and data structures then about the functional organisation at the start of a project.

Data-structure oriented methods focus on data structure rather than data flows. Examples for these methods are represented by JSD, Jackson System Development, by M.A. Jackson [24] and DSSD, Data Structured Systems Development, by J.D. Warner and K.T. Orr [25]. Each of these methods has a distinct approach and specific diagramming techniques. The common characteristics of such methods is, that they identify key information objects (also called entities or items) and operations (also called actions or processes). Each method assumes that the structure of information is hierarchical. Each provides a set of steps for mapping a hierarchical data structure into a program structure.

Data modelling concepts have been introduced into data-flow oriented methods in recent years. The Entity-Relationship model proposed by P.P Chen [26] is mainly used. General aspects of data modellings have been discussed by S.M. Fisher [3]. Examples of the graphical representation, Entity-Relationship Diagrams, have been shown in Figs. 12 and 13.

Data contents and data structures are generally recorded in a data dictionary. This contains definitions for all data on diagrams, in functional specifications, in the code, or in the data dictionary itself. This data dictionary acts as a unique source of all information on data. A computer readable data dictionary is invaluable to ensure consistency and completeness of data across all phases and products of the software life cycle.

9. MANAGEMENT PROCEDURES

Management procedures play an essential role in the overall software development process. In order to conduct a successful development project one must understand the scope of the work to be done, the resources in manpower required, the tasks to be accomplished, the milestones to be tracked, the cost to be expended, and schedules to be followed. Software project planning provides this understanding. Useful techniques for cost and schedule estimation and tracking of the estimates against actually achieved progress have been developed in the past years. Very detailed discussions of basic principles and procedures for software project planning, implementation planning, quality assurance, testing strategies, and configuration management, can be found in many books on software engineering [8, 9, 12].

Strategies for the use of such procedures are either defined by using a particular software development methodology (chapter 11) or are decided for a given project or a whole organisation. An example of this is provided by ESA [27]. Figure 18 shows the overall life cycle management scheme that has to be followed for every software project developed in-house, by contractors, or software companies.
Figure 18 Software life cycle management scheme

Figure 19 provides an overview of the software life cycle management procedures to be followed during various phases of the development.
10. **AUTOMATED TOOLS**

Nearly all activities associated with software production have been manual in the past. However, many of those activities could, and should, be replaced by computer supported facilities. Currently tremendous investments are being made to develop automated tools for all phases of the life cycle and to support various methodologies. The clear aim is to improve productivity and quality of the software being developed. The key concept is to maintain all information in a central data base which can be accessed by anybody who wants or needs information at any time.

Automated 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 data base is essential to go back and forth between information relating to various phases of the project.

In the past 2-3 years there have been dramatic changes due to the availability of reasonably priced workstation hardware. Development of data bases and sophisticated user interfaces which exploit features of the new hardware allow creation of well integrated sets of software development tools which have now appeared on the market. This may finally provide the real breakthrough for a wider application of software development methodologies.

Rather few CASE tool packages were available commercially for the support of the full SA-SD methodology in 1984. Several more supported only the Analysis phase and extensions for the Design phase were just promised. Data modeling was not available then. This was even worse for other methodologies which had virtually no support tools. The situation has changed dramatically since then. A recent survey of CASE products and vendors lists 135 entries [28] and amongst those some 31 products available for the support of development of real-time systems [29]. These numbers do not include tools traditionally used in the coding phase (compilers, debuggers, source control, static analyzers, etc.)

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.

11. **SOFTWARE METHODOLOGIES**

The coherent combination of methods, management procedures, and automated tools is
called Software Development Methodology. Unfortunately there exists no single methodology which could be equally well applied to all projects or in all environments (e.g. software development in a University or Research environment will be quite different from a defense contractors environment). Some of the methodologies are better suited for a particular field of applications (e.g. embedded or real-time systems, business systems, transaction processing, telecommunications, scientific or engineering projects).

Surveys can provide very useful information for the selection of a methodology suitable for a given field of application and environment. A very thorough comparative survey [30] had been used for our own selection. Table 1 shows a condensed summary for the applicability of various methodologies available in 1982. A more recent survey can be found in [28].

### Table 1
Methodology applicability

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Embedded</th>
<th>Sci/Eng</th>
<th>O/S</th>
<th>Tools</th>
<th>DP/DB</th>
<th>Expert/Alg</th>
<th>Sm</th>
<th>Med</th>
<th>Lg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM/PCM</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DADES</td>
<td>S</td>
<td>I</td>
<td>I</td>
<td>S</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DSSD</td>
<td>W</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DSSD</td>
<td>W</td>
<td>W</td>
<td>S</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>EDM</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>GEIS</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td>E</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>HOS</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>IBM/PSD-SEP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISEM</td>
<td>S</td>
<td>I</td>
<td>S</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ISAC</td>
<td>I</td>
<td>W</td>
<td>S</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>JSO</td>
<td>W</td>
<td>S/W</td>
<td>W</td>
<td>W</td>
<td>E</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>MERISE</td>
<td>W</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>NIAM</td>
<td>E</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>PRADOS</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>REMORA</td>
<td>W</td>
<td>S</td>
<td>I</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SA/SD</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SD</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>S</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SDM</td>
<td>S</td>
<td>S</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SEPN</td>
<td>W</td>
<td>S</td>
<td>I</td>
<td>S</td>
<td>S</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>SREM</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>STRADIS</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>I</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>USE</td>
<td>I</td>
<td>W</td>
<td>I</td>
<td>W</td>
<td>W</td>
<td>S</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

**Key:**
- W = well suited
- S = satisfactory
- I = inappropriate
- X = suitable
- (W or S not specified)
- IE = insufficient experience
- ... = no answer
- $S$ = "Large is 250K lines, 75-100 effort years"
- blank = is not suitable

Sci/Eng = Scientific/Engineering
O/S = Operating Systems
DP/DB = Data processing, database
AI = Artificial intelligence
12. **PRACTICAL EXPERIENCE**

One can find many papers in the literature which relate (mainly positive) experiences with using the techniques discussed. In this section we would like to sum up our own practical experience in using SA-SD for about three years for several projects. Very similar conclusions have been drawn by other groups in High Energy Physics who have used SA-SD for a variety of projects (ALEPH, SPS and LEP control systems, Fastbus system management, OPAL, D0, H1, Zeus, Cleo).

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 much more coherent product despite the many people, with their individual 'tastes', who contribute to it,
- the idea to spend a sizable 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 a better product at the time of startup of production,
- the partitioning of work for several teams working independently is useful in our current context where software development has to be done distributed geographically,
- 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,
- the methods are flexible enough to be adapted to specific needs and it is reasonably easy to learn and use them.

Inconveniences or problems encountered:
- one of the most serious problems was non-availability of adequate tools - either because they did not exist yet or were too expensive or required expensive hardware platforms. This situation has changed now and a number of suitable CASE tools are being offered,
- 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 applications, the methods used during the Design seem to be much more tedious and labour-intensive since they can not so easily be supported via automated 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. **OUTLOOK TO THE FUTURE**

Software Engineering techniques have been successfully transferred from the research area into the real world and are now available in the form of comprehensive methodologies which are taught in seminars and are supported by automated tools. Many of the CASE tools have now some of the extensions needed for real-time design.

However, at present only few of these tools can handle concurrency and parallelism well enough to move smoothly from design to implementation. There are still gaps to move from DFD/CFD and STD to the details needed for actual coding. Extension of the tools will allow better automation of code generation and better simulation capabilities and several activities are under way. Amongst these a group including P. Ward, co-author of the real-time extensions now widely available [13], is trying to expand and combine existing notations to provide a more complete design discipline proposing ESML, the Extended System Modeling Language [31]. Very recently methods of real-time structured analysis have been adapted to support a combined hardware/ software specification and design for an application specific system on a single chip [32]. Several CASE tools are now available with sophisticated support for programming languages like ADA, C, and C++ in the near future.

Though the driving force behind these rapid developments is the US aerospace and military establishment the methods are now also widely used outside this area for systems programming, telecommunications, on-line transaction processing, fault-tolerant data processing, robotics and others.

The methodologies are well suited for development of new software systems. However, a maybe surprising fact is that about 80% of all programmers are working on maintenance and upgrades of existing software and do not design software starting 'from scratch'. The basic problem is that one wants to preserve the investment in manpower, knowledge and resources that went into the development of the software. Productivity for rewriting is very poor since one generally has to derive actions and intents from existing code with incomplete or no documentation available. There is therefore strong demand for tools for reverse-engineering of existing code. This would allow to restructure and document existing programs and help the programmer to translate source-level code into specification-level descriptions. These can then provide the basis for an enhanced version, where existing CASE tools can be successfully used. Obviously this task is not easy but several tools are starting to appear [33].

14. **CONCLUSIONS**

Software Engineering techniques have been used within ALEPH and other groups in High Energy Physics for approximately three years for a variety of projects. The methods have been found to be extremely helpful in providing a clear understanding of the required functionality of the system and as a means of communicating ideas and suggestions both within the groups and to the Collaboration as a whole. Lack of convenient automated tools in the past has meant that SA-SD has been used more as a collection of useful techniques for producing reliable and
robust software rather than a rigorous methodology for proceeding from a requirements specification to a validated set of program modules. This situation has changed now and we expect that the methods will be used more widely in future.

*Should one use these methods NOW?* YES - even if there are new developments and new tools. One should not forget that the software stone-age ended just some 30 years ago! Many of the basic concepts will not change much. It is best to start with these methods as soon as possible since it has been convincingly shown that the software creation process is much improved. Adopting these new methods implies a major change of personal habits, changes to the methods can always be incorporated later.

*Which methodology to use?* This depends on the characteristics of the projects as mentioned above. SA-SD seems to be applied successfully to a wide variety of projects. It is also well supported by several CASE tools.

*Where should the methods be applied?* Definitely for medium to large projects where many people participate in development, projects of long life-time, and projects with an important evolution phase. Initially it is much better to use them for small projects since one passes much faster through the various phases and gets a better feeling which methods are well suited and which are not in the particular context. One should not forget the 'learning curve' - it takes about 2-3 projects before one is really familiar enough with the methods in order to improve also efficiency and productivity.

*Who should learn the methods?* All people who are involved in analysis and design of software. All people who code and test the software (if not the same people as above), but maybe they need to be less familiar with all details. The managers who are responsible for the overall project (hardware and software) - this is the best way that they can monitor progress of the development process.

*The important point is to start using a consistent way to produce software.*

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, W.Zhao, J.Knobloch, M.Green, J.Bunn, J.Harvey, R.McClatchey, T.Charity, A.Putzer, the ADAMO team in general and F.Dydak in particular for his foresight.

* * *

REFERENCES

C. Gane, T. Sarson, Structured Systems Analysis, Prentice-Hall (1979),


[20] D. Parnas et al., copies of several papers in Naval Research Laboratory Memorandum Reports, Washington, D.C. received from author.


