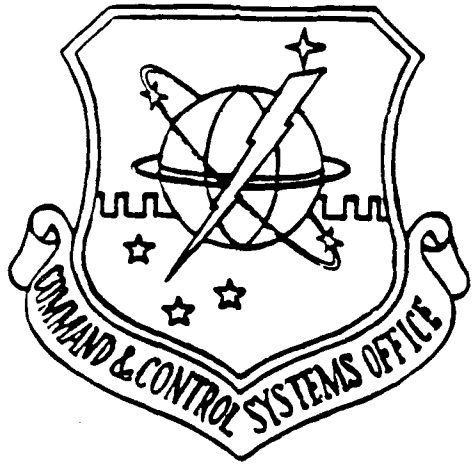
ADA* EVALUATION PROJECT

AN ARCHITECTURAL APPROACH TO DEVELOPING ADA* SOFTWARE SYSTEMS

Prepared for

HEADQUARTERS UNITED STATES AIR FORCE
Assistant Chief of Staff for Information System
Technology & Security Division



Prepared by
Standard Automated Remote to AUTODIN Host (SARAH) Branch
COMMAND AND CONTROL SYSTEMS OFFICE (CCSO)
Tinker Air Force Base
Oklahoma City, OK 73145-6340
AUTOVON 884-2457/5152

*Ada is a registered trademark of the U.S. Government
(Ada Joint Program Office)

21May 1986

THIS REPORT IS THE SECOND OF A SERIES WHICH DOCUMENT THE LESSONS LEARNED IN THE USE OF ADA IN A COMMUNICATIONS ENVIRONMENT.

ABSTRACT

This paper reports on the impact of the Ada environment and philosophy on the software design process.

The report first describes problems with the currently accepted software lifecycle model, and then proposes a different model. The recommended Spiral Model of Software Development is based on work done by TRW Corporation.

The next section describes the process used to select design methodologies. The fundamental goals are to produce software systems that are adaptable, reliable, and understandable -- and to produce them in an efficient manner.

After problems are discussed with the currently used and available design methodologies, an alternative design approach is described. This Architectural Model of Software Development is a multi-dimensional, or multiple view, approach using several distinct methodologies.

Finally the paper gives an example of applying both the Spiral Model and the Architectural Model to a software development project. The paper states that both the Spiral Model and the Architectural Model are not in themselves methodologies; Instead, they provide the framework for applying existing software engineering principles. This paper will, hopefully, stimulate interest in developing an automated, full lifecycle methodology based on some of the ideas presented.

Ada Evaluation Report Series by CCSO

Ada Training	March 13, 1986
Design Issues	May 21, 1986
Security	May 23, 1986
Module Reuse	Summer 86
Micro Compilers	Fall 86
Ada Environments	Fall 86
Transportability	Winter 86-87
Modifiability	Winter 86-87
Runtime Execution	Winter 86-87
Testing	Spring 87
Project Management	Spring 87
Summary	Spring 87

TABLE OF CONTENTS

Appendices

L I S T O F F I G U R E S

1. INTRODUCTION

1.1. BACKGROUND

This paper documents the work and thought processes done in choosing ways to design an Ada language project -- a project that has a tight schedule. The resulting software system must work well.

So that potential Ada developers could gain a practical insight into what was required to successfully develop Ada software, the Air Staff tasked the Command and Control Systems Office (CCSO) with evaluating the Ada language while developing real-time digital communications software. The evaluation was to consist of a number of evaluation papers, one of which was to deal with design. CCSO chose the Standard Automated Remote to AUTODIN (Automatic Digital Network) Host (SARAH) project as the basis for this evaluation.

SARAH is a small to medium size project (approx. 40,000 lines of source code) which will function as a standard intelligent terminal for AUTODIN users and will be used to help eliminate punched cards and paper tape as transmit/receive media. The development environment for SARAH consists of the SOFTECH Ada Language System (ALS) hosted on a Digital Equipment Corporation VAX 11/780, ALSYS Ada compiler for the IBM PC-AT, a Burroughs XE550 Megaframe and several IBM compatible PC-XT and PC-AT microcomputers. The ALS environment is the focal point of this integrated development environment. The source code developed on the XE550 and microcomputer workstations is maintained by the ALS configuration control system and will be transferred to the PC-ATs for final compilation and targeting. The SARAH software targets are the IBM compatible PC-AT and PC-XT microcomputers.

Because the system must be reliable, maintainable, and reusable it was decided to use modern software engineering concepts and methodologies to the greatest extent possible. The draft Software Volume of the Air Force Information Systems Architecture also recommends the use of Ada and formal software engineering methodologies in the development of Air Force software systems.

The Department of Defense (DoD) language that best enabled implementation of software engineering concepts was Ada. What remained to be chosen was what design methodology would be best suited to analyze and design the system. As we will describe, this decision was not easy.

1.2. PURPOSE

The purpose of this paper is to share our experiences with the

rest of the Ada community so others may learn from them. If we can provide some ideas that help to further the work being done on design methodologies, so much the better.

1.3. ASSUMPTIONS AND CONSTRAINTS

The assumptions and constraints are as follows:

- One possible constraint is the size of the SARAH project. Since the SARAH project team is small (10 persons), and since it will only be some 40,000 lines of code, some of the experiences reported in this paper may not be appropriate for larger groups and larger projects.

- The evaluation is based on a limited time budget for writing the paper. More research could be done in the area.

- The SARAH team members have a variety of previous experience. Some members have had very little software experience. Others are very experienced in system design and have a good working knowledge of some design methodologies.

- Since the SARAH project is at the analysis/design stage of development, the final effectiveness of the concepts presented in this paper cannot be fully evaluated at this time. At the completion of the SARAH project, a summary paper will reflect on how well the design approach worked.

2. LIFECYCLE DEVELOPMENT MODELS

When managing and controlling the development of software it is necessary to conceptualize the process of development. This process of abstraction is necessary for the human mind to grasp the essence of complicated physical processes. Indeed, software development is one of the most complex endeavors attempted by society. Over the years there have been a number of different models for understanding the development and post development support of software. The model that has received widest support is the waterfall model. There have existed some other models such as:

- Stagewise Model[14]
- Parnas Information Hiding Approach[15]
- The Twoleg Model[16]
- Evolutionary Model[17]
- Automation paradigm[18]

Of this group of classical models, the waterfall model has been most widely accepted in the Department of Defense. We will, therefore, limit our discussion of older models to the waterfall model.

2.1. WATERFALL MODEL

The waterfall model has been the most common approach for addressing the entire software lifecycle in Air Force wide policy and documentation. For example, the new DOD-STD-2167 is based upon this approach to building software. The typical graphic representation of this model is shown in Figure 2-1.

The waterfall approach has been used successfully for a large number of projects; however, the model has often been criticized, particularly by the Ada community. Some of the criticisms include:

- no provision for quick prototyping
- no mechanism for risk analysis
- inappropriate reviews
- does not effectively support modern software engineering practices

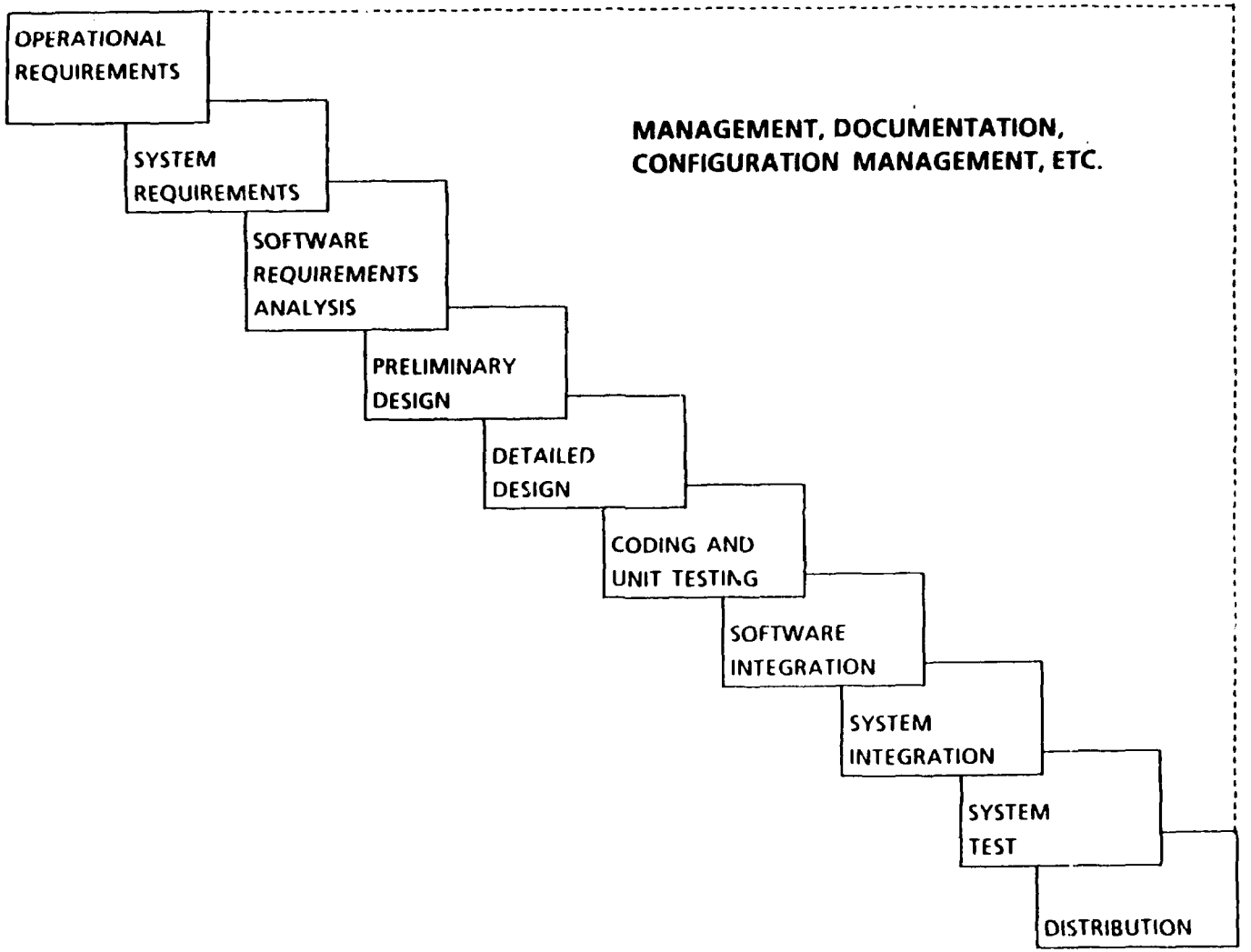


Figure 2-1. Waterfall Model

Each of these suggested problems will be discussed:

For software projects where the risk of an incorrect user interface will seriously jeopardize the usefulness of the software, a quick prototype should be developed prior to full scale development. Ada provides the features necessary for quick prototyping; however, the waterfall model does not provide a flexible framework for this type of development. Using the waterfall model, it is necessary many times to build the system twice before the entire problem is fully understood. During the analysis phase, the end user is often unclear of the exact system requirements and the developer may have his own (different) perception of how the system should work. Problems in specification may not be discovered until the integration and testing phases. The developers must then go back to the analysis phase to introduce additional or changed requirements.

The pure top-down or waterfall process does not allow a group to look at high risk issues until late in the development process. Generally, some risk analysis is conducted at the major reviews, but this is not formalized. In large projects, a great amount of effort may have been needlessly expended prior to the review. This can greatly raise the risk of the entire project. Although a look-ahead step can be introduced into the model, resources cannot be easily directed to high risk areas so that further development does not continue until the major risks are properly assessed. In essence, with the waterfall model, all development must progress through each phase in parallel.

The reviews associated with the waterfall have often been criticized as being more appropriate for hardware development than for software. A major review in a waterfall approach is the Critical Design Review (CDR). The CDR marks the beginning of the coding phase; however, most of the important decisions are generally made well before this review. For example, with Ada, the overall system architecture and interfaces are of major importance. If the high level design is well defined, the detailed design and coding phases become largely mechanical. This is even more true if the software is developed with a large number of reusable components. Reviews should be structured so the most important elements of the design are reviewed prior to applying the bulk of the development resources.

The waterfall model does not provide the flexibility to allow for modern software engineering practices such as software component reuse and automated development. Software component reuse should be considered during the analysis and design phases. To enhance productivity, a software designer must be aware of the software components available so that the systems requirements can be structured to make use of these components. As such, the analysis and design phases become more integrated and much less defined than those depicted in the waterfall model. Moreover,

with the introduction of knowledge based software development tools, a model with defined analysis and design phases may no longer be applicable.

The waterfall model can be used effectively in some situations, but it is not appropriate for the development of all software. Consideration needs to be given to the relative risk of the development, and the components and environment in which the software project is to be done. An even more generic model may be needed so that various approaches can be integrated into the overall development model. Fortunately, some very good work has recently been done in the area of lifecycle models. A new model that we found to be excellent will now be discussed.

2.2. SPIRAL MODEL

As discussed above, software developers require a more flexible lifecycle model than is currently provided by the waterfall model. The spiral model[5] provides this flexibility. This model was developed by Barry Boehm (TRW Defense Systems Group) and is beginning to gain a high degree of acceptance from within the software community. The graphical representation of the model is shown in Figure 2-2.

In the graphic of the model, the radial dimension represents the added incremental cost incurred in completing the developmental steps. The angular dimension represents the progress made in completing each cycle of the spiral. The basic premise of the model is that a certain sequence of steps is repeated while developing or maintaining a software system. The steps are first done at a very high level of abstraction, then each loop of the spiral represents a repeat of the steps at successively lower levels of abstraction.

SPIRAL MODEL

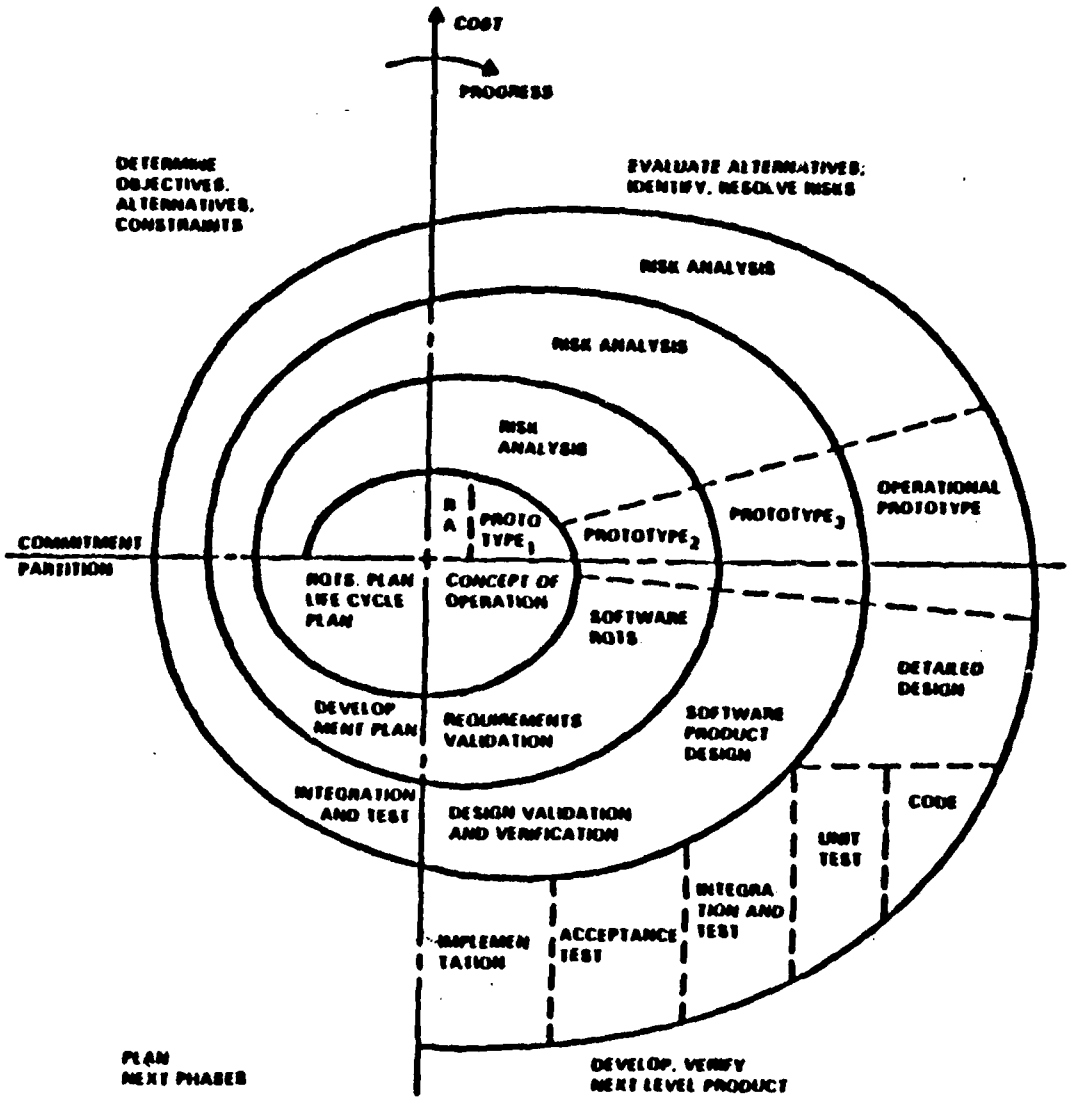


Figure 2-2. Spiral Model

The spiral model is appealing for a number of reasons. First, there is a definite planned examination of project risk at each major abstraction level. Second, the model accommodates any mixture of specification oriented, process oriented, object oriented, simulation oriented, or other approaches to software development. In certain situations, it can reduce down to one of the other more common models. For example, the waterfall model is an important special case for the development of systems where user interface and performance risks are minimal. The model provides a flexible framework for developing software systems using modern software engineering techniques, automated environments,and advanced programming languages such as Ada.

There are, however, a number of areas which should be addressed before the spiral model can be considered as a standard model for software development. These are:

- the model does not match the current world of government contract software acquisition.
- the model places a great deal of reliance on the ability of software developers to identify and manage sources of project risk.
- the spiral model process steps need further elaboration to ensure that all of the participants in a software project are operating in a uniform and consistent context.

Hopefully, future work done by groups such as STARS (Software Technology for Adaptable, Reliable Software -- a DoD initiative) and the Software Engineering Institute will consider some of the problems associated with the spiral model and provide specific guidelines on how the model can be applied in a number of different cases.

2.3. SARAH LIFECYCLE DEVELOPMENT MODEL

SARAH development was somewhat constrained by the MIL-STD 2167 documentation standard. Since the standard is based on the waterfall model, the SARAH designers needed to ensure that the development approach was compatible with the specified phases of development, reviews, and documents. Although this made it more difficult to make use of the Spiral model, it was possible to apply a spiral approach within this framework. Risk analysis was performed throughout the design and an evolutionary approach was used for the high risk sections of the project. A detailed account of how the spiral model was used for the SARAH project is included in a later section of this paper.

To summarize, although current documentation standards for government projects are based on a waterfall approach (i.e. DoD-STD-2167), considerable benefits can be gained by implementing some of the spiral model features within this framework. If the

full benefit of advanced software development technology is to be realized, a large amount of work needs to be done in the area of lifecycle development models and associated documentation standards. The waterfall model should be considered only as a special case within the spiral framework. The developer should be able to choose from a range of other applicable models so software production can progress at a satisfactory rate, with relevant reviews, and with sufficient risk analysis and management.

3. CONSIDERATIONS FOR CHOOSING A DEVELOPMENT APPROACH

In addition to a good lifecycle model as described in the last section, it is necessary to choose a model and/or methodologies to use for the design phase of the lifecycle.

The complexity level of today's software is increasing at an alarming rate because of the ever increasing requirements for more advanced systems. Development and support of software for major systems is one of the most complex human endeavors, often requiring hundreds of people for five or more years at costs exceeding $100M (for example, the B-1B program)[22]. This level of complexity is beginning to exceed that manageable by humans without special tools to assist in managing the complexity. Some of the observable symptoms of trying to create and manage these highly complex software systems without the proper tools are severe time and cost overruns and extremely high costs for post development software support.

A partial solution for this problem of complexity is to use a development methodology that helps the system designers manage complexity, better understand the software system, and better communicate that understanding to users, programmers, and software maintainers. In fact, the draft Software Volume of the Air Force Information Systems Architecture specifies that such methodologies should be used[13].

3.1. METHODMAN STUDY

When the DoD was planning for the implementation of the Ada language and environment, they knew that well defined design methods would have to be used for the development of Ada software.

Although several methodologies were then being used for the design and development of software, there had been no comparative studies for these methods. There was a lack of information on how applicable these methods would be for the development of Ada software systems. In addition, although some software organizations were applying design methods to software development, a large proportion of the industry was unfamiliar with this approach. The DoD subsequently sponsored methodology research to promote the use of methodologies for Ada software development.

The DoD sponsored methodology study was named Methodman[10]. The first draft of the Methodman document was completed by Peter Freeman and Anthony Wasserman in November 1982 and consisted of the following three papers:

- Ada Methodologies: Concepts and Requirements
- Ada Methodology Questionnaire Summary

- Comparing Software Design Methods for Ada: A Study Plan

The Methodman study has made a significant contribution to the software community. Methodman provides a comprehensive set of requirements and evaluation criteria for selecting and creating development methodologies.

3.2. METHODOLOGY REQUIREMENTS

The Methodman study states that a development methodology should support the four fundamental goals of software engineering: modifiability, efficiency, reliability, and understandability. Since requirements change over the life of a software system, the development methodology must provide maintainers with the flexibility to easily change the design and implementation. In addition, if the design is well structured and flexible, the performance and storage characteristics of the system can be more easily "fine-tuned" to enhance efficiency. Another major requirement for a software system, and hence the development methodology, is understandability. If a design is not understandable, then maintenance and development costs will be high. There have been many instances where time and cost overruns have been experienced because the developers could not communicate their design to users. If a design is understandable, then problems can be detected early in the development process.

Freeman and Wasserman expand these basic requirements to include a number of specific requirements. Methodman indicates that a development methodology should:

- cover the entire developmental process
- enhance communications between users and developers
- support problem analysis and understanding
- support top down and bottom up development
- support software validation and verification
- support the development organization
- support system evolution
- provide automated support aids
- be teachable and transferable
- be open ended

A development methodology should also directly support software

reuse. A major problem experienced by software developers is productivity. One method of increasing productivity is to use reusable software components. Currently, most software is being generated in a line by line fashion and a large amount of the available programmer resources are being used to redevelop software that had already been developed for other systems. This is necessary because the languages used for the older systems are machine dependent and the software modules are very dependent on one another. The Ada language provides features such as generics and packages which can be used to produce reusable modules. However, if the development methodology does not support reuse, then these reusable elements will be difficult to implement into a design.

An important requirement for methodologies is that they allow for automated tool support. Productivity can be significantly improved through the use of automated support aids for analysis, design, and maintenance. A great deal of attention has been given to automating functions such as configuration management; however, there have been few attempts to provide Computer Aided Design (CAD) for Software. A major reason for this is that there are currently few development methods which can be effectively supported by automated aids. Most of the current development methodologies were not created with automated support as a requirement and so most attempts at providing CAD for these methodologies have been less than satisfactory.

A development methodology should provide the means to effectively manage complexity. The methodology should provide the designer with a means of abstracting the solution so that only the features relevant to a particular abstraction level need be addressed at a certain point in time. For example, consider the design of an electronic mail system. At the highest level of abstraction the designer knows that the system must be able to communicate with a packet switching network. Other considerations at this high level of abstraction may include a message preparation facility, a mail directory system, and a display manager. However, deciding how the software will physically address the communications port would introduce additional and unnecessary complexity at this level of abstraction. The methodology must allow the designer to layer a design so that the complexity of a particular layer does not become uncontrollable.

3.3. EVALUATION CRITERIA

We have covered some of the major requirements for a development methodology but what should we look for when choosing a methodology for a particular project? When evaluating methodologies, look for:

- support of functional hierarchy
- support for data hierarchy

- o defined interfaces
- o control flow
- o data flow
- o data and procedural abstraction
- o support for concurrency

The methodology should not be planar; rather it should support data and functional hierarchy. The methodology should provide a mechanism for showing data and control flows at different levels of abstraction.[6] In addition, a depth or hierarchical view should be provided so that module dependencies and system structure can be shown.

The methodology should not constrain the software designer to one type of abstraction or programming paradigm. Most current methodologies support only one major form of abstraction. As such, all aspects of the solution must be viewed from a single perspective. Real world solutions require the designer to consider both functional and object groupings. For example, if a car designer was forced to only take an object oriented approach to design, he would be able to identify the major objects such as the engine, drive train, steering, and braking systems. However, some form of procedural or process abstraction would have to be applied to determine how all these components function together. Also, the designer needs to look at concurrent processes. For example, will the braking system operate in parallel with the engine operation or will the brake system require the engine to be running before the braking system is effective?

Similarly, a software system consists of a number of objects which work together to provide a defined functionality. In addition to procedural and data (or type) abstractions, a methodology for Ada should provide a concurrency abstraction mechanism.

To summarize, the interfaces between software modules should be well defined and show data flow, control flow, and synchronization. Well defined interfaces promote modularity and component reuse. In addition, if the interfaces are well defined, development time can be reduced because the time taken for system integration and testing will be less. Maintenance costs will also be lower because the maintainers will have a better understanding of the system and will be able to more easily determine the overall effects of software modification.

4. DEVELOPMENT METHODS FOR ADA

The use of a development methodology for the production of Ada software systems is extremely important. As software systems become larger and more complex, some method must be used to manage software development. To manage complexity, the software development methods must support software engineering principles and goals. For example, the principles of abstraction and information hiding allow developers to concentrate on the overall system architecture without being distracted by the lower level implementation features. Since Ada was based on modern software engineering principles, major benefits can be gained if the language features are used to construct software systems. A development methodology for Ada should therefore allow designers to implement solutions which make good use of these features.

4.1. METHODMAN METHODOLOGIES

As stated in the previous section, the Methodman study provided evaluation criteria for selecting Ada applicable methodologies, and a summary of 24 design and development methodologies that could be used for developing Ada software. Of the methodologies listed in Methodman, the following have gained widest support:

- Structured Analysis Design Technique (SADT)[21]
- Jackson System Development (JSD)[8]
- Structured Analysis Structured Design (SA/SD)[20]

These methodologies were not developed with Ada in mind. As such, they provide a generic design which could be coded in a number of different languages. One of the disadvantages of this approach is that the methodology may not take advantage of special features available in a particular language. In fact, this has happened with Ada. Many of the Ada abstract concepts that support software engineering principles are not considered as part of the design methodologies.

Of the Methodman methodologies, JSD is seen as the most applicable for Ada development. JSD provides good lifecycle coverage from analysis to maintenance and provides an effective mechanism for designing real-time concurrent applications. However, one of the major criticisms of JSD is that it is a proprietary methodology so there are very few training courses available. In addition, the diagrams produced do not easily allow designers to communicate designs to users.

Structured Analysis/Structured Design has been very popular for developing software systems implemented with older languages such as Pascal and COBOL; however, the Structured Design portion of the methodology does not effectively allow designers to use the more advanced Ada features such as tasks and packages. Although

Structured Design is not effective for Ada development, several organizations are using the Structured Analysis element of SA/SD to analyze system requirements. This analysis phase is generally followed by a design phase which uses an Ada design methodology. If this approach is used, the transformation from analysis to design needs to be addressed in detail. Many times the thinking that resulted in the creation of the Data Flow Diagrams (i.e. analysis) is very "cold" when it comes time to map the ideas into the real-world implementation of Structure Charts (i.e. design). Our experience has shown that this step can be difficult.

4.2. ADA SPECIFIC METHODOLOGIES

Since the release of the Methodman document, several other methodologies have been developed. Of these, the following have been the most popular:

- Object Oriented Design (OOD)[7]
- Process Abstraction Methodology for Embedded Large Applications (PAMELA)[11]

These methodologies are Ada specific and so make direct use of the language features.

OOD has been very popular for developing Ada applications. In fact, OOD has received a kind of "cult" following. OOD provides a very elegant mechanism for data abstraction and provides a "cook book" approach for design. Indeed, the relative ease of application has been a major reason for OOD's popularity. The OOD methodology is based largely on the principle of data abstraction. Data abstraction is very important for system development since it reduces the amount of coupling between modules and helps manage complexity. However, OOD does not readily support process abstraction and so is not very effective for developing real-time concurrent applications. In addition, OOD covers only the design phase of the software lifecycle and so some other mechanism must be applied, such as Structured Analysis, for the analysis phase. As discussed, the transition from analysis to design could be difficult if this approach is used. In summary, OOD provides several excellent design features; however, OOD should not be adopted as the "eternal cure-all" for software design.

The PAMELA methodology was designed to allow for the development of large, real-time embedded systems. Based largely on the principle of process abstraction, PAMELA has been used successfully by a number of organizations to develop real-time Ada systems. PAMELA is a relatively new methodology and as such has been prone to a large number of changes. Some of PAMALA's positive aspects include its applicability for automation and the easy transition between analysis and design. However, PAMELA does not effectively support a bottom up approach to development and does not support an effective mechanism for data abstraction.

4.3. SELECTING A METHODOLOGY

There is currently no single methodology that satisfies the needs of all software development projects. Several of the methodologies that had almost become defacto standards in some areas are now proving to be incomplete. For example, many groups had adopted classical SA/SD or SADT as the ultimate solution for the analysis and design phases. However, newer thinking and newer language capabilities, spurred largely by activity in the Ada language community, showed that these methodologies could not effectively support the design of systems using advanced language features. As such, the Ada community has moved towards Ada specific methodologies. These methodologies provide many of the features needed to develop Ada software systems; however, there is currently no one methodology that is applicable for the development of all Ada applications.

The approach taken for the SARAH project was to study the existing methodologies and select a methodology which would best serve SARAH development. Since the SARAH requirements called for a multi-tasking system, some degree of process abstraction was required. To accomplish this, the SARAH designers decided that features from PAMELA, JSD, and Buhr[12] would be needed to implement the concurrent and process oriented elements of the system. In addition, an object oriented approach was required for grouping the major functions and data elements. This was required for a number of reasons. First, software reuse was an important consideration and this approach would more easily allow existing software to be integrated into the system, and SARAH software to be used on later projects. Second, since the SARAH system would process sensitive data, carefully controlled data abstraction was needed to protect data within the system.

Armed with the system requirements and the requirements for the development methodology, we established the guidelines for developing the SARAH system. We found that none of the existing methodologies would accommodate the methodology requirements. As such, the relevant features were taken from a number of methodologies and these were integrated into a multi-view development model. The resulting model is presented later as the Architectural Model of Software Design.

To summarize, the research, development, and refinement of methodologies is not yet complete, or even mature. Even so, the methodologies that are available can be used effectively to develop superior software systems which are transportable, maintainable, and reliable. In the future, many more methodologies will be developed and these will be supported by intelligent Computer Aided Design workstations. Organizations need to keep current with methodology research so that new advanced methods can be integrated into the overall software development strategy as they become available.

5. THE ARCHITECTURAL MODEL OF SOFTWARE DESIGN

When examining the existing methodologies and design representations, we found that none of them corresponded all that well to the way system designers seemed to approach a software design. In order to represent on paper (or computer screen) an accurate representation of the software design process, what we needed was a different model.

5.1. MULTIPLE VIEWS

We have a traditional saying that "a picture is worth a thousand words". When we examine the engineering disciplines, we find that they all make extensive use of graphical representations (abstractions) of the problem, or project, on which they are working. This is because they have found that the human mind can, indeed, assimilate the abstract concept represented by a picture must faster than trying to use words.

5.1.1. UNDERSTANDING

Taking the concept a bit further, two pictures should be worth even more, and so on -- at least up to a reasonable point. If we look at the work of an Architectural Engineer, we will find that if the object trying to be described is the design of a house, we will be provided with several graphical representations of that object. One such abstraction of a house is shown in Figure 5-1.

We can see that it certainly gives us some information about what the resultant house will be like -- but, it does not give us a complete understanding. Were we in the market for a home, we would probably not give the go-ahead to build based on this one graphical representation. If we combine our knowledge gained from the first picture with the information in Figure 5-2, we have a much better idea of what we will get.

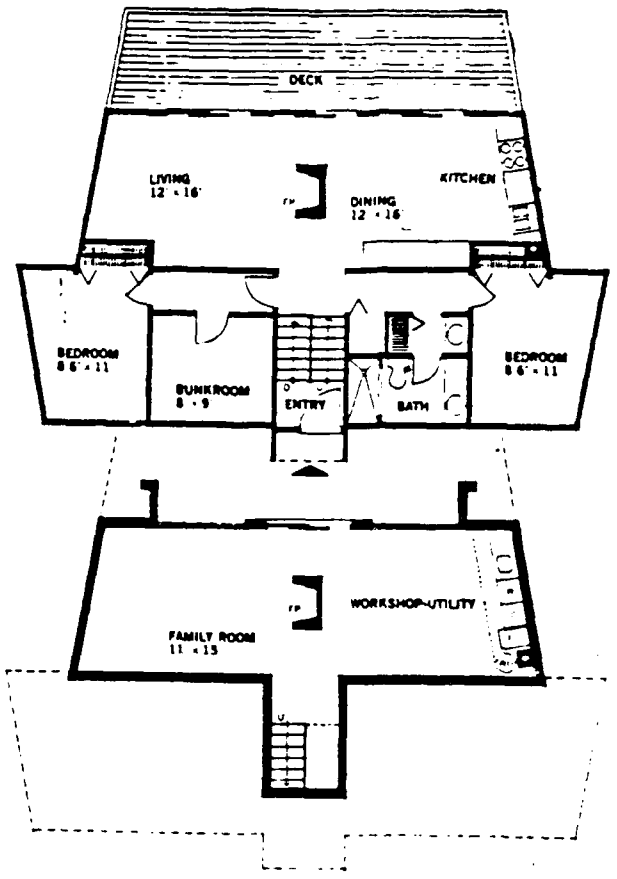


Figure 5-1. Floor Plan

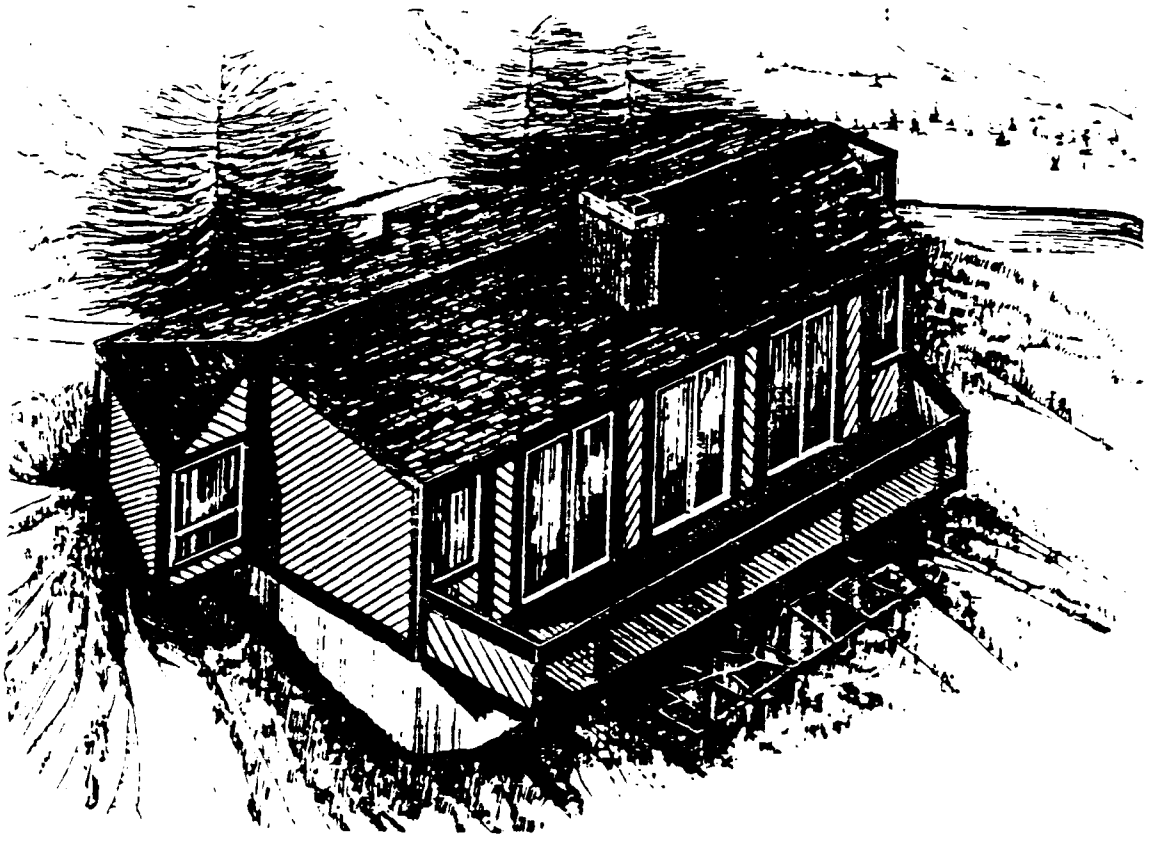


Figure 5-2. Artist's View

5.1.2. COMPLEXITY

Another reason an Architect will use multiple views of a structure is to manage the problem of complexity. Even the graphical representation of a house would be unwieldy if a single picture had to show everything about the structure. We typically find that the architect prepares a "user" view (elevation) that is a drawing of what the house will look like from the outside. Another user view (floor plan) will show how the living space is organized.

The Architect will also prepare a graphical representation of the plumbing for the house, and a separate graphical representation for the electrical wiring. These pictures allow the plumbing and electrical contractors to efficiently do their jobs. Each can look at the appropriate plan, and quickly comprehend what is needed -- all without having to sort through data that is not relevant to what they need to do.

5.2. ARCHITECTURAL VIEW OF SOFTWARE DESIGN

When we examine most computer software design methodologies, we see that virtually all of the accepted methodologies use at least one form of graphical representation of the design. This graphical representation is typically joined with some narrative text to further enhance the reader's understanding of the concepts.

We have found that when we want to fully describe a software design -- especially when we want to design in software engineering concepts to be implemented in Ada -- no single design representation was adequate.

We also found that our system designers tended to have several different, but concurrent, abstract views of the system they were designing. After some examination of what was happening, we have come to the conclusion that their analytical process was most similar to that of the structural architect described above. Instead of having the elevation for the user view, and having plumbing and electrical plans, the designers had abstract concepts for the menu system (a user view), the connectivity (or topography), the concurrency (where applicable), and the structure (or hierarchy) of the system they were designing. What was missing from the design process was an overall structure in which to explicitly acknowledge the existence of these abstract views, and workable graphical representations for them.

Thus our alternate, or architectural, view of system design involves multiple abstract views of the system being designed. Figure 5-3 is a graphical representation of the Architectural Model.

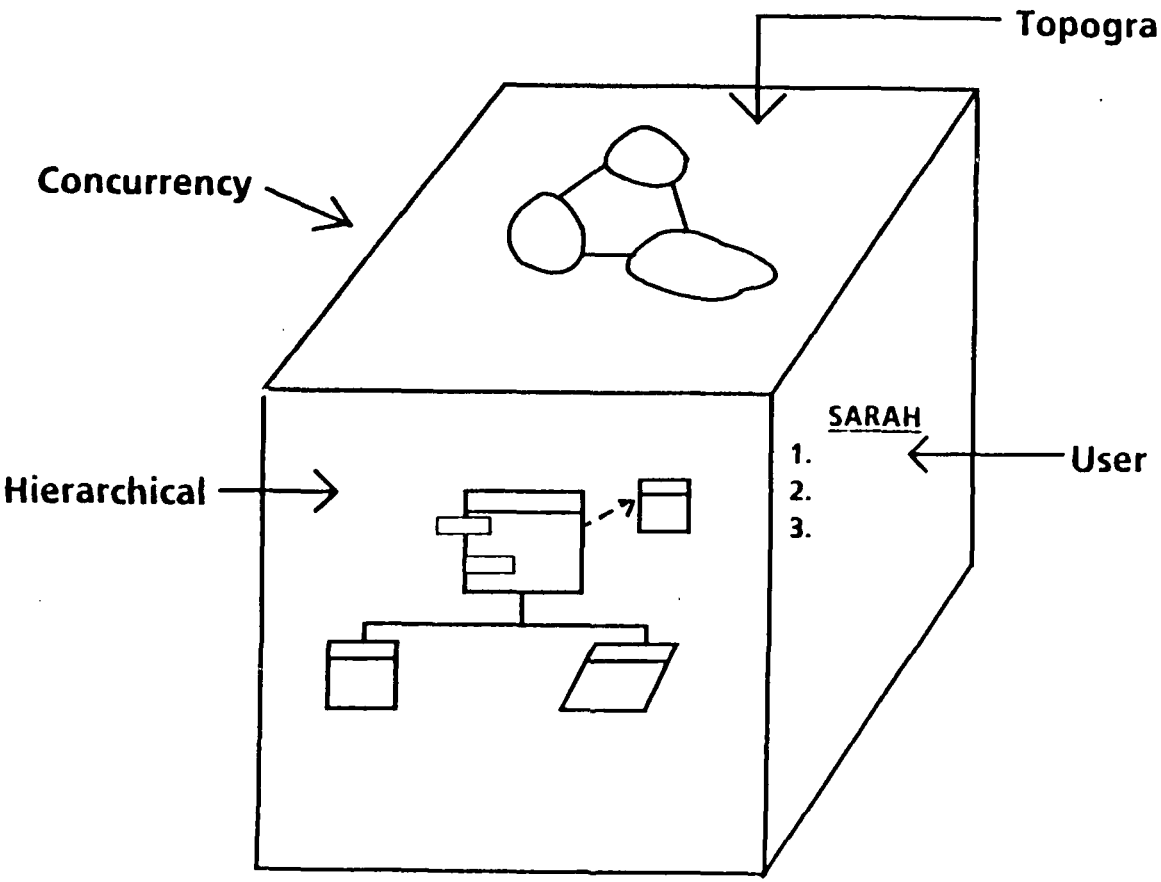


Figure 5-3. Multi - View Model

Each of the four views of the system places emphasis on different qualities and characteristics of the system, and has meaning when viewed alone. However, the real value is the synergistic effect of using all the views together to understand and represent the system. Additionally, we have found that being able to explicitly examine one view at a time greatly helps the management of complexity.

In one sense, using the Architectural Model of Software Design can be likened to watching a sculptor at work. After shaping the object a little, the sculptor will look at it from the side. Seeing some change that is needed, more sculpting will be done. Then a look at the other side or maybe the top is needed. Then, a little more sculpting is done. Then, another look, etc.

In order to understand the significance of each of the four views, more needs to be said about each of them.

5.2.1. TOPOGRAPHICAL VIEW

First, we will take the cube of Figure 5-3 and rotate it so that we can view the face that shows us the Topographical view. The topography is a representation showing what the major components of the system are, and how they are connected. This abstraction is not so much concerned with the physical structure and dependencies of the software units; it is primarily focusing on functionality, and the major relationships between.

The primary goals in showing a topographical representation of the system are:

- Show the most important functional and data groupings in the system at the current level of abstraction.
- Create a graphical representation of the system that will be understandable by people not currently involved in the design (e.g. users, software maintainers, etc.).
- Manage system understanding complexity by separating one of the abstract ways in which we think about software systems.

Figure 5-4 shows the cube rotated so that the topographical view is visible to us. The notation shown is basically a number of "clouds" that represent our understanding of major groupings of either logical functions or data and objects.

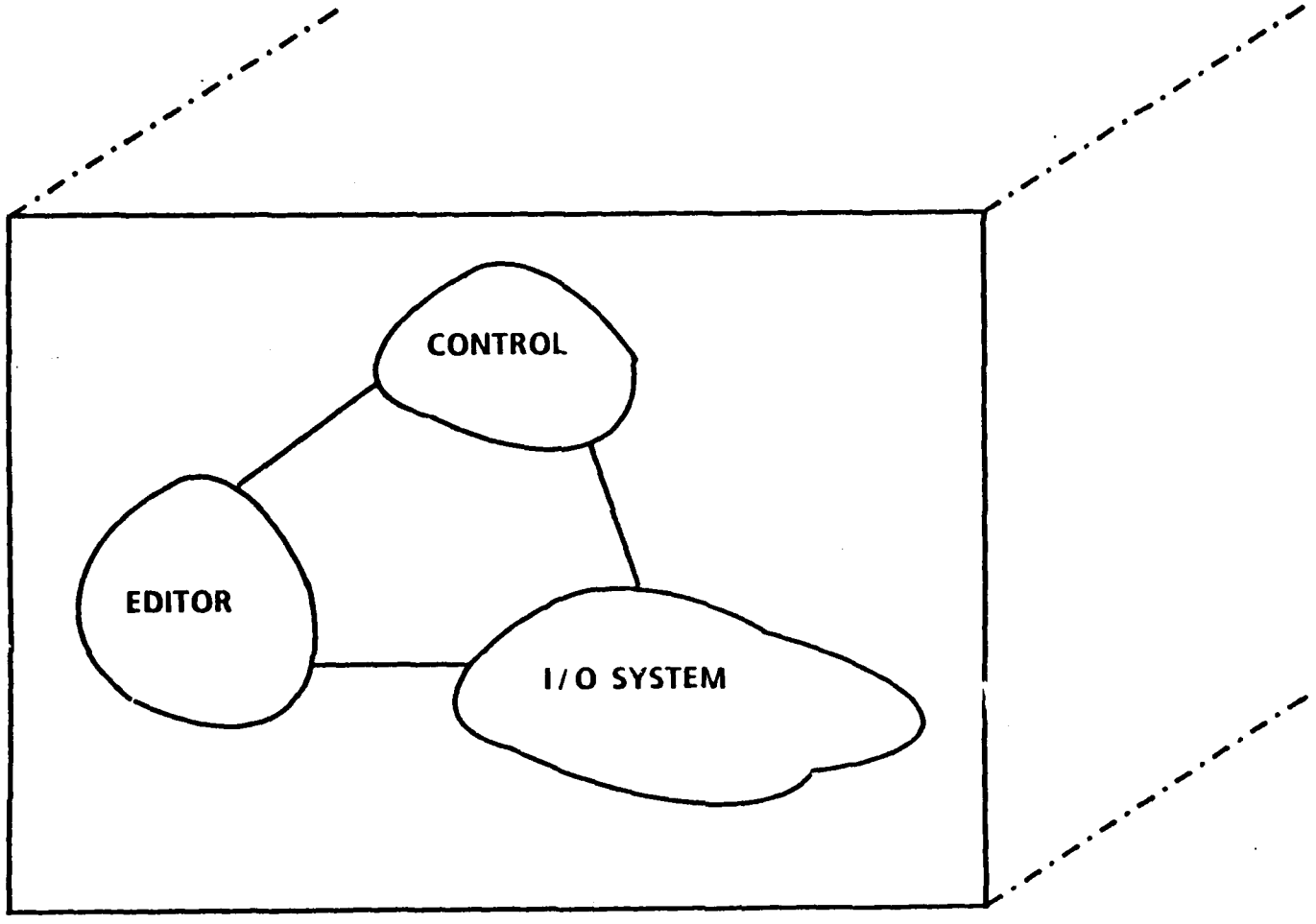


Figure 5-4. **TOPOGRAPHICAL VIEW**

This type of abstraction borrows heavily from the basic concepts behind using the classical Data Flow Diagrams. There are some basic differences:

- The topographical notation is not strictly limited to functional decomposition and representation of data transforms; it may show groupings by data and object considerations.
- It can show control where appropriate.
- It can be later combined with the knowledge gained from the hierarchical view to create a view of the system that shows not only a logical abstraction, but a physical representation of the system being created.

The topographical representation is similar to the classical Data Flow Diagram (DFD) in that one of the logical ways to first analyze a system is to identify major functions and data flows between them. Therefore, this well-accepted type of abstraction is of great value.

One of the primary things to note about the topographical representation is that it is generally a very flexible abstraction. The general rules we use are simply:

- Limit the number of clouds to less than seven.
- Try to limit the number of data and control lines shown to less than about fifteen.

There are no rules about what may be in a cloud, or what a line between clouds must mean. We simply are using this picture as a way to understand the system, and to convey that understanding to other people on the system development team.

We feel we can responsibly have a relaxed rule set because the topographical view is not the only graphical representation of the system -- we have three other graphical representations to help us understand the system, and in which to share the complexity of understanding the system.

5.2.2. USER VIEW

We will now rotate our design cube to look at the user view. Figure 5-5 is a representation of doing this. You will note that there appears to be a system menu shown on the face of the cube. The reason a menu was chosen was because the most common user view of a software system is the menu.

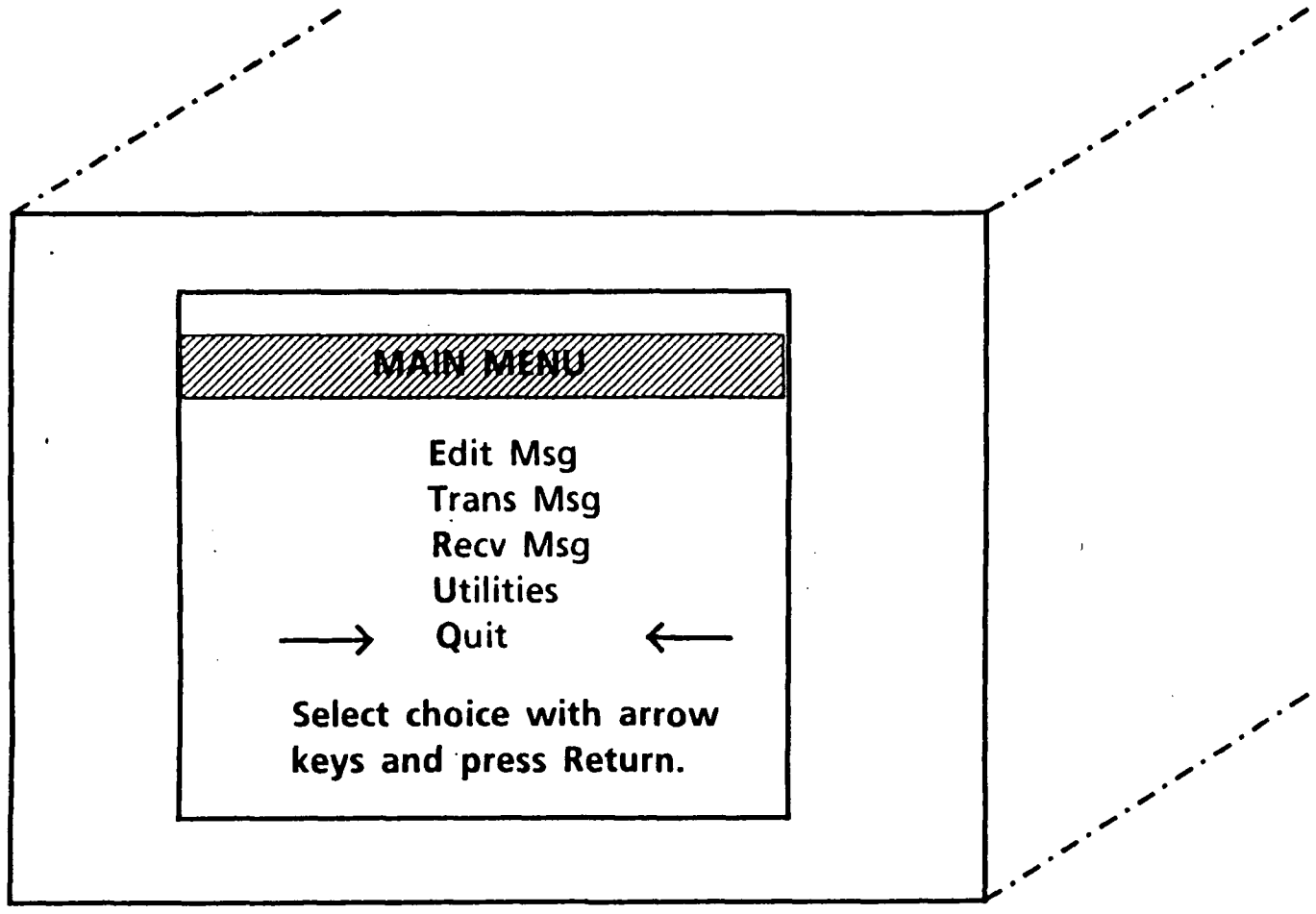


Figure 5-5. USER VIEW

The purposes of the User View in the Architectural Model are to:

- Force the system designers to examine and design the human interface.
- Help understand the system by having an explicit representation of the user view.
- Manage system understanding complexity by separating one of the abstract ways in which we think about software systems.

Most of the existing software development methodologies do not consider the user view of the system as an integral part of the design process. The human interface is most times tacked on to the system after all else has been designed (and many times coded and tested). To this approach we strongly disagree. The way in which a software system interacts with the user can have dramatic influences on how that system is structured and organized (i.e. designed).

During the earlier days of our profession we had our hands full just trying to get a handle on how to manage the complexity of getting the basic functional processes done. There was certainly at least a subconscious desire to not make the situation more complex by adding yet another thing to consider while figuring out how a system was to be built.

With the Architectural Model, we find that having the many separate graphical representations allows us to adequately consider the user view of the system without complicating the other parts of the system design. In fact, we have found that, quite to the contrary, the design of the user interface early in the design process actually enhances our intuitive understanding of the system. This actually helps so much with the other aspects of the system design that the time spent on the user view is more than saved by the shortened times spent on the other portions of the design.

The above considerations are just from the point of view of putting together a system that functionally works, and that is well organized, reliable, and maintainable. If we want to consider how well the system is accepted by the end user, then we must put some hard work into the user interface. A good analogy might be the auto industry. The "art" group comes up with an auto design they believe will be pleasing to the eye of the buyer. They also try to arrange the instrumentation, etc. so that it will be easy to see and use. The engineers try their best to provide all the functionality specified by the "art" group. They also try to use all the state of the art mechanical innovations, and fit them all into the package done by the "art" group. Sometimes they can't make it fit, so the "art" folks make changes to accommodate the engineers. Thus, the engineering aspects and the artistic design aspects of an automobile do

affect each other during the early design. So it is also with software.

Consider an automobile company that refused to think about what a car would look like until after the frame, engine, drive train, etc. had already been put together? They probably wouldn't sell many.

The user view should consider and show what the system will look like to the user. This may be in the form of defining menus, screens, windows, or help messages. We have found that actually drawing "screen" on paper, and then filling it in to pictorially show what the user would see,to be very workable. It is interesting to see how the team's understanding of the system is developed when they are forced to "operate" the system by paging through paper "menus" and "screens".

An additional benefit of having this graphical representation is the ability to communicate with the future users of the system on a level that they can really understand. In the past, showing a user a data flow diagram or structure chart might well have impressed them that we software people were "smart" and that building software was "magic". Why not show them what the screen will look like when they get the system? They will certainly identify missing or misunderstood requirements faster from seeing menus and screens than from other "technical" graphical representations of the system.

5.2.3. CONCURRENCY VIEW

As we rotate our multiple view cube yet another time, we see the face labeled Concurrency View (Figure 5-6). The notation shown on the cube face may not be as familiar as those previously discussed.

The goals of the concurrency view are:

- Show in a graphical representation which processes can run concurrently.
- Show any dependencies related to concurrency. For example, process "A" can run concurrently with process "X"; but, process "X" must be running before process "A" can run.
- Provide an explicit representation, forum for discussion, and means to communicate to others the relevant concurrency factors for the system.
- Manage system understanding complexity by separating one of the abstract ways in which we think about software systems.

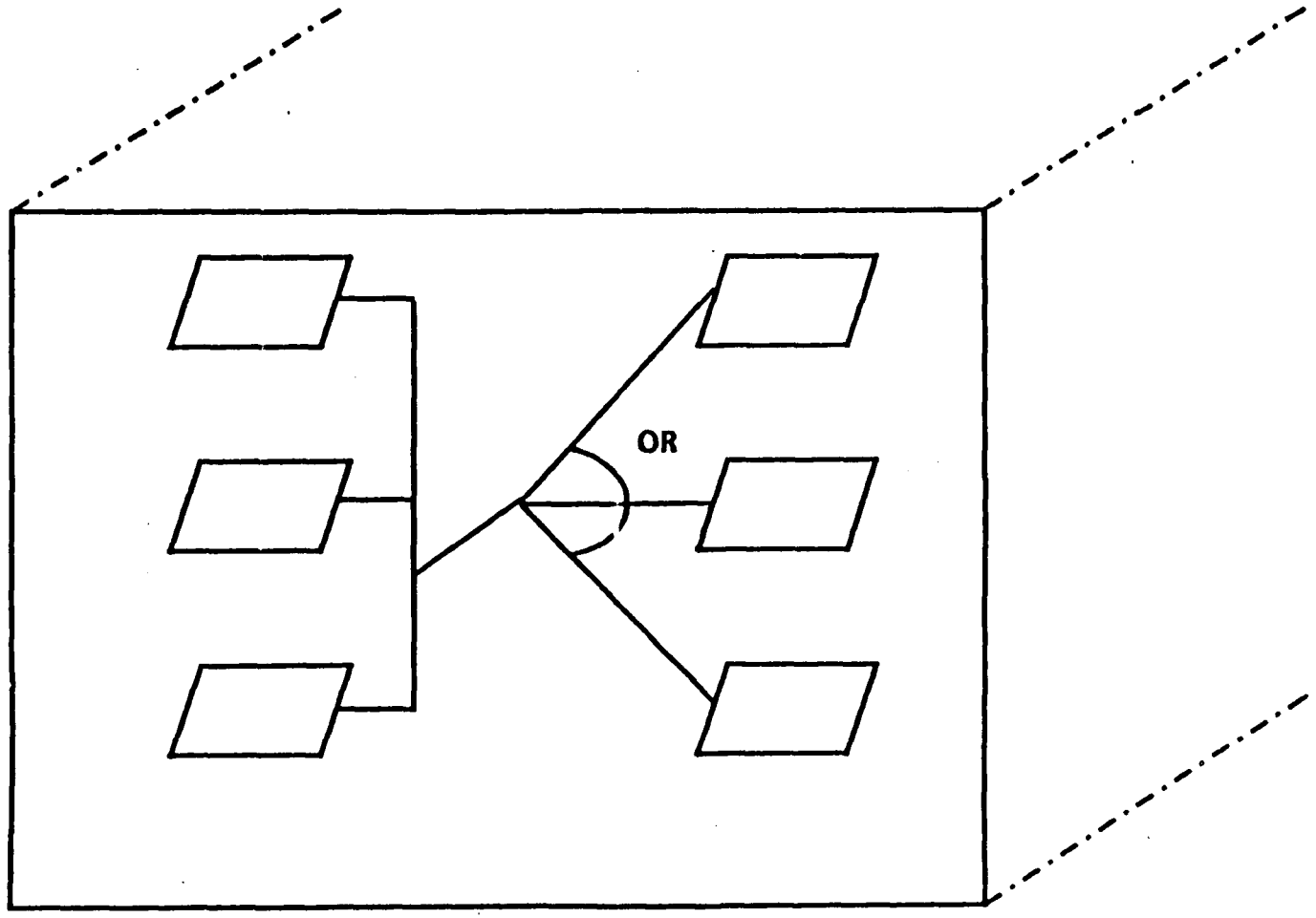


Figure 5-6. CONCURENCY VIEW

The traditional design models do not allow the designer to annotate concurrency at different levels of abstraction. Indeed, it is another one of those things that a designer would have to "keep in the back of his head" while documenting the system using most other design models.

Most traditional business applications have had no need to design concurrent operations. Because concurrency usually meant assembly language programming at a level very close to the hardware (i.e. complexity and non-transportability), no one would consider concurrent process unless it was an absolute requirement. With the advent of many multiprocessor computers and a DoD standard Ada language, that situation is changing. Even when the functional system requirements do not mandate concurrency, the system designer must consider using concurrent operations to enhance system efficiency, system modularity, system reliability, and maintainability.

On the example of Figure 5-6 the vertical line on the left is the concurrency line. Anything connected to it with a perpendicular line is considered on the concurrency line. That is, those processes that operate concurrently. The lines connected to the concurrency line at angles other than ninety degrees represent processes that can become concurrent. Visually, you can imagine the line pivoting so that it lays on top of the concurrency line. It can also pivot back out and again become non-concurrent. Second level dependencies are shown in a similar fashion with angled lines joined to a second level concurrency line.

5.2.4. HIERARCHICAL VIEW

If we rotate the Architectural Model cube to the fourth face, we see the hierarchical view. This view is shown in Figure 5-7. This view of the system represents the organization, packaging and structure of the software.

The goals of using this view and its graphical representation are to:

- Show module dependencies.
- Show visibility of program units and data.
- Provide a tool that enhances a structured way of discussing and representing the mapping of the design into the Ada language.
- Provide a graphical representation that will help communicate the system structure to secondary design groups and programmers.
- Manage system understanding complexity by separating one of the abstract ways in which we think about software systems.

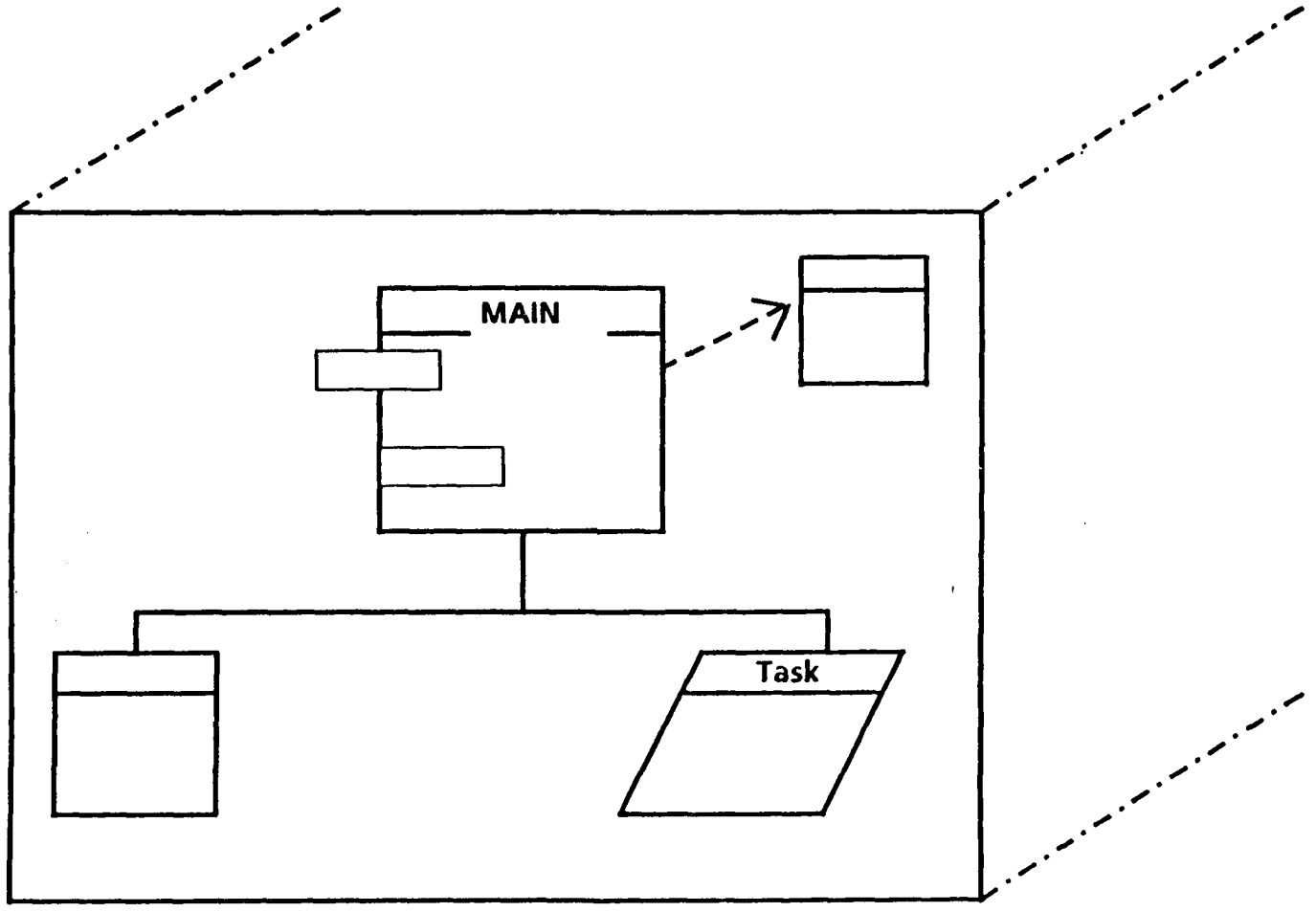


Figure 5-7. HIERARCHICAL VIEW

Although there is some similarity between this abstract representation and the invocation or structure charts of classical Structured Design (SD), there are also significant differences:

- SD structure charts have only two basic types of program unit notation. Those mapped into the two types of program units found in most of the existing high order languages -- main program, or sub-programs. The Ada language has a much richer selection of basic program units (e.g. packages, tasks, generics, etc.). It is important to indicate not only the hierarchy of the system, but also the type of program unit, and the visibility.

- The traditional SA/SD approach dictates a complete functional decomposition (using data flow diagrams), and then an abrupt transition to design (using structure charts). Most people demonstrate a very difficult time making this transition. With the constructs and program units available in Ada, coupled with proper structure notation, the design of the system program and data hierarchy can be done at each of the levels of abstraction along with the other graphical representations of the Architectural Model.

The notation used for the system hierarchy is based upon work done by Burkhardt and Lee published in Ada Letters[9]. Since their work was very well thought out, we quickly replaced our more primitive notation.

The main features are that internal program units (procedures, tasks, generics, etc.) can be shown in the parent program unit if little detail is needed, or they can be shown as separate expanded units below the parent when appropriate. With dotted lines showing the "withing" process added to the basic structure, it is easy to visually examine and evaluate data and program visibility and dependencies.

To summarize, our experience has shown that it is helpful to have both the topographical and the hierarchical charts visible to the design team at the same time. Although all the views are often used and have their own relative merit, these two are by far the most used.

6. APPLYING THE MULTIPLE VIEW MODEL TO SOFTWARE DEVELOPMENT

In order to enhance the understanding of the concepts we have been presenting, it should be beneficial to go through the steps used in the actual design of the SARAH communications workstation.

In order to have some rough idea of what SARAH would be, it was decided to first make a "mini" loop through the Spiral Model. This would allow some data and thoughts to be organized enough to allow a "go-ahead" decision to be made.

6.1. SPIRAL LOOP 1

As with most projects, the very first thing that must be done is to develop the rough idea for the software system into a viable concept of operations that can be used as a basis for further discussions. Another reason for doing some initial analysis and thinking on the project is to gather enough ideas and information so that risks and potential problems can be evaluated before the decision is made to expend more resources.

6.1.1. CONTEXT

As a vehicle for discussion, and as an initial attempt to graphically portray the overall context of the SARAH system, the design group created the diagram in Figure 6-1. This graphic shows the software system in the central box, and shows the major interfaces and data flow to devices that are external to the SARAH software. If other existing software systems were involved, they would also be shown on this chart.

This external interconnect graphic is a one-time exercise. As the system design progresses through lower and lower levels of abstraction, the external interconnect situation does not change. Therefore, there are no lower levels of this system representation. In essence, the central box labeled SARAH is what is designed when the decision is made to go forward with full scale design and development.

It is important to note that the graphic itself does not quite go far enough to communicate the overall system concepts. Thus, a narrative description of each of the data lines, and for each box on the chart was also done. A representative example of this narrative description is shown in Figure 6-2.

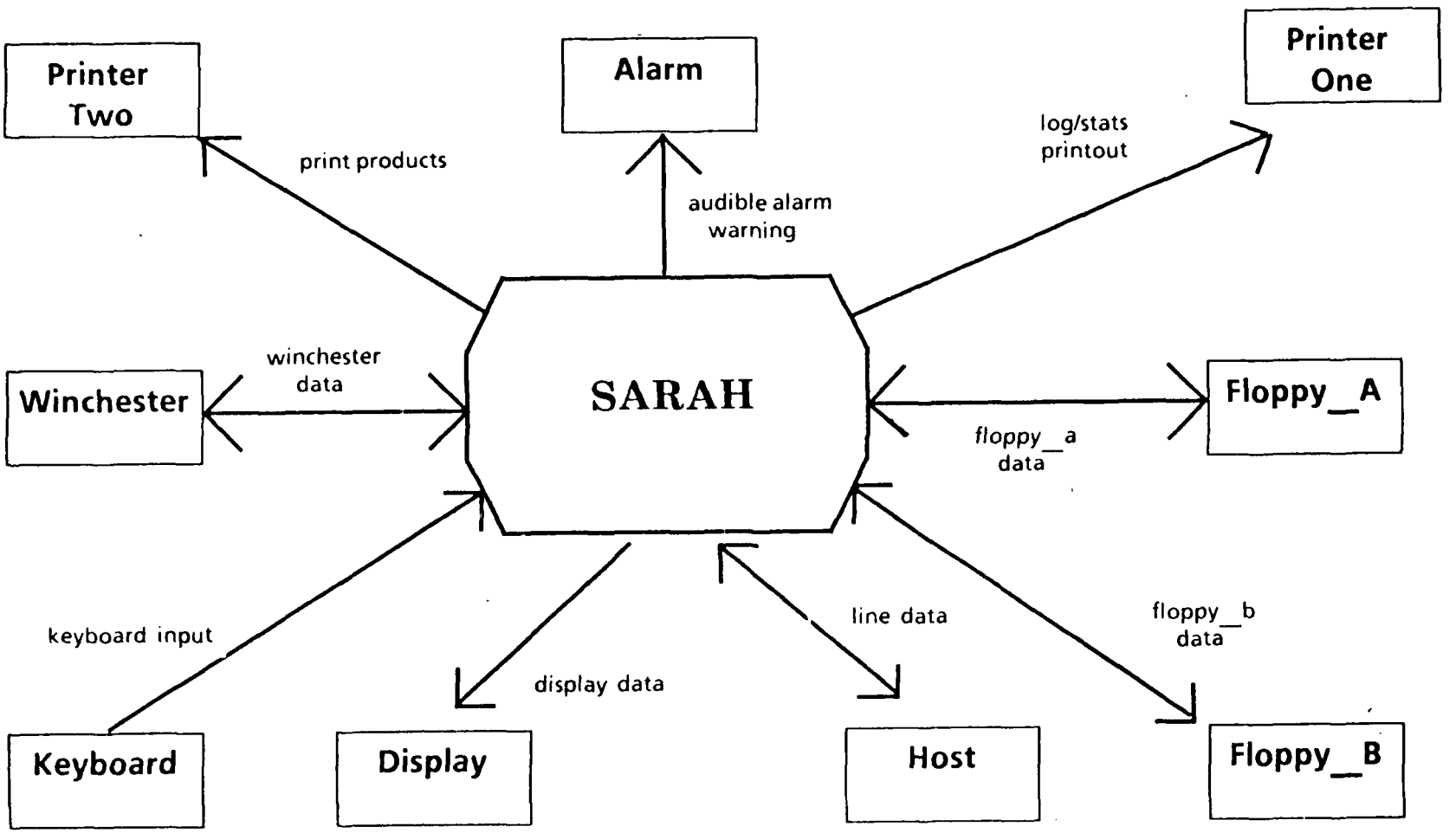


Figure 6-1. SARAH EXTERNAL INTERCONNECT

SARAH Workstation Physical Devices Con't

8. Display:

-- Display can be a monochrome or color monitor. If a color
-- monitor is used, a Color Graphics Adapter will be
-- required.

9. Host:

-- Host can be AFAMPE, Mode I, or IS/A AMPE. The host must be
-- able to speak Mode I or the SARAH asychronous protocol.

The Data To/From SARAH Physical Devices

1. Floppy_B_Data:

-- Floopy_B_Data will consist of transmitted messages, the SARAH
-- Floppy Table of Contents, updates to the SARAH Floppy Table of
-- Contents, and a hidden diskette validity indicator.
-- Floppy_B_Data may also be any Floppy_A_Data data.

2. Floppy_A_Data:

-- Floppy_A_Data will normally consist of ASCII text files,
-- wordprocessing files, the SARAH Floppy Table of Contents,
-- hidden diskette validity indicator. Floppy_A_Data may also be
-- any Floppy_B_Data.

3. Log/Stats:

-- The Log_Stats printouts consist of any of the system
-- statistics or message log information collected for the SARAH
-- workstation. Log/stats printout may be divided into three
-- categories: Receive Message Log, Transmission Message Log, and
-- System Statistics and Messages.
--
-- (a.) The Receive_Message_Log is used to record information
-- about messages received from AUTODIN or a host processor.
-- It consists of the internal SARAH message number, the time
-- received, the message header, the file time and the delivery
-- destination.
--
-- (b). The Transmission Message Log is used to record
-- information about messages transmitted from the SARAH
-- workstation. It consists of the internal message number,
-- the time sent, the message header, the text file name on the
-- SARAH input diskette.

Figure 6-2. Narrative for SARAH External Interconnect

6.1.2. RISK ANALYSIS

The resulting representation of the SARAH system is the initial basis for making a risk assessment for the project. The primary risks identified were:

- Procurement lag time for both hardware and Ada compilers needed for the project.
- Performance issues with the target hardware (Zenith Z-150 and other IBM PC compatible machines) -- doing concurrent communications and other work such as text editing might be too slow on this class of machine.
- Long learning curve for the Ada language and environment.
- Possibly excessive time needed to accomplish DoD standard documentation for a relatively small project.

The initial graphic representation of the overall SARAH system was very helpful in promoting an overall understanding of the scale of the project.

6.2. SPIRAL LOOP 2

After the commitment was made to proceed with full scale design and development of SARAH, then it was time to start into the second loop of the Spiral Model. Within this loop we planned to apply the Architectural Model at the highest abstraction level.

Upon completion of the design of the highest abstraction level, the Ada coding for the system structure should be possible. While being used to test the high level software system organization as visualized in the design, this code can also serve as the structure for an early prototype.

It was decided that a prototype showing at least the user interface and high level menus would be invaluable in further refining requirements with the user. It might also give management some visibility into what the project group is doing, providing them with some assurance that what comes out at the end of the project is what they want. The prototype would end the second loop through the Spiral. Then risks would again be assessed, and any adjustments to the schedule, the design, or the overall approach could be made before starting the next loop of the spiral to go down another level of abstraction.

6.2.1. USER VIEW

One of the first views to be considered was the user view. Not only did we feel it was important to have the system functionally correct, we wanted the system to be easily usable by both novice and experienced users.

It might be noted here that although this paper is presenting the four different views of the Architectural Model one at a time, the design and development team would many times be looking at several of the views at the same time. That is the value of having the different abstractions: One can look at any one on its own, or look at two or more together to see the correspondence in design.

A sample of the documentation generated for the user view is shown in Figures 6-3 and 6-4. Some of the highlights of the graphic in Figure 6-3 are:

- It indicates that a general windowing approach is to be used for the user interface. This was picked because it was judged to be the most user friendly (i.e. intuitively workable) means of managing a screen.

- The picture shows that there will be several major types of windows used, and it describes their purpose.

- A help window is described. This will ensure that help information is generated while the system design is done. The "wait until the last minute" help system might not be done as well as one considered throughout the life of the design.

- The "Note Window" is the primary abstraction to handle concurrent process communication with the operator. For instance, while the operator is doing text editing, the communications task could need to send a "note" to the operator. This would appear in the note window. It would not immediately overwrite the text work being done on the main part of the screen. The operator could acknowledge the note by simply pressing the F9 key, which is dedicated to note acknowledgment. The text work on the screen is not disturbed.

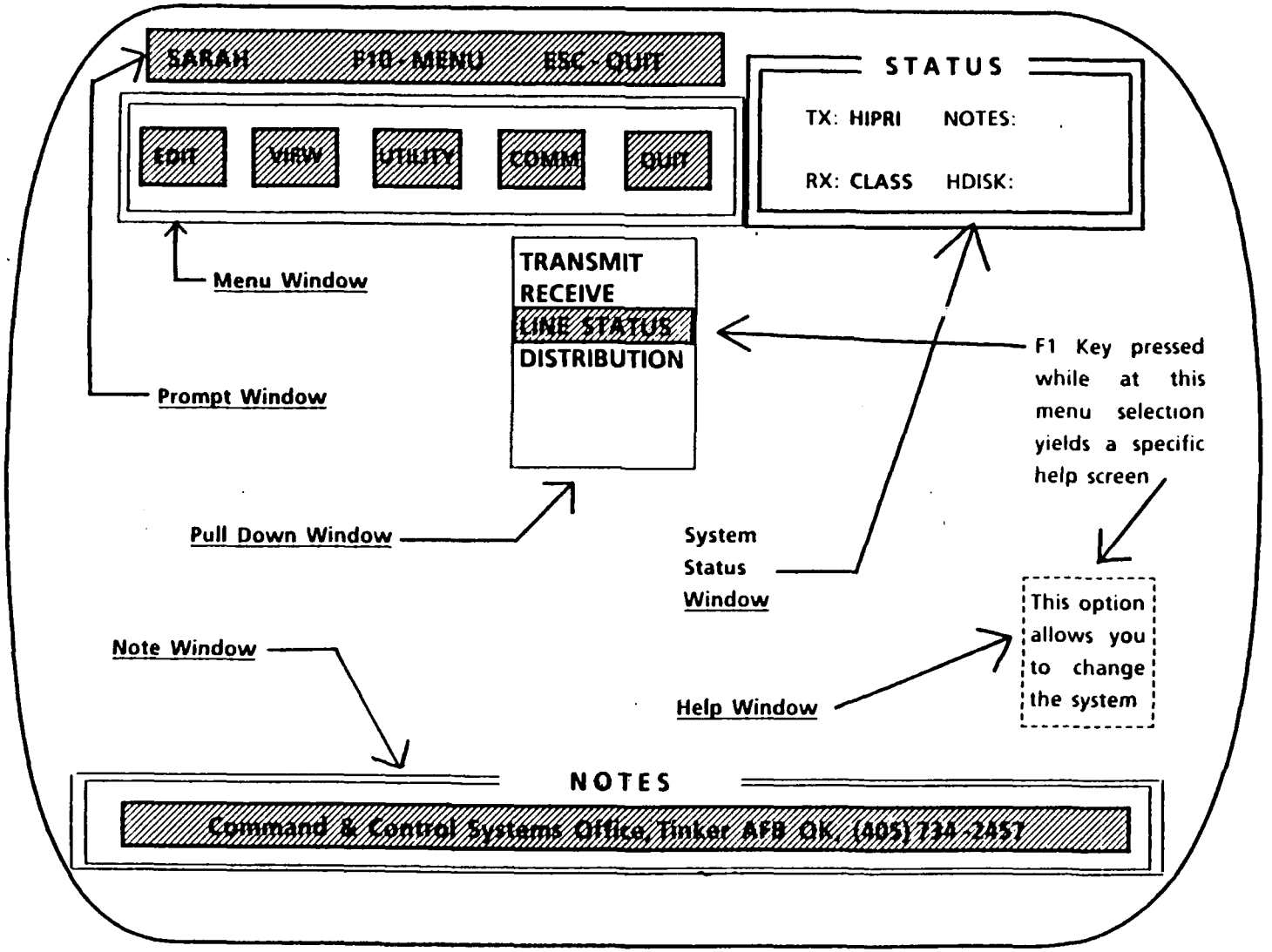


Figure 6-3. **User View. Window Naming Conventions**

Figure 6-4 is a lower level user view of the SARAH system. It shows a lower level menu selection (the DD Form 173 option within the Edit menu). The approach taken by the designers to document the system was to create the help text for all the menu options. Each menu selection would have its own sheet of paper showing the corresponding help window(s).

This process serves several purposes:

1) It forces the designers to thoroughly think through the system requirements (i.e. what the system is to do).

2) It provides a form of documentation that is readily understandable by the future user of the system.

3) It ensures that the help information is developed early in the design so that it can be "fine tuned" throughout the design process to be most meaningful to the user.

4) It aids in communicating an understanding of the system design to people involved in the project development.

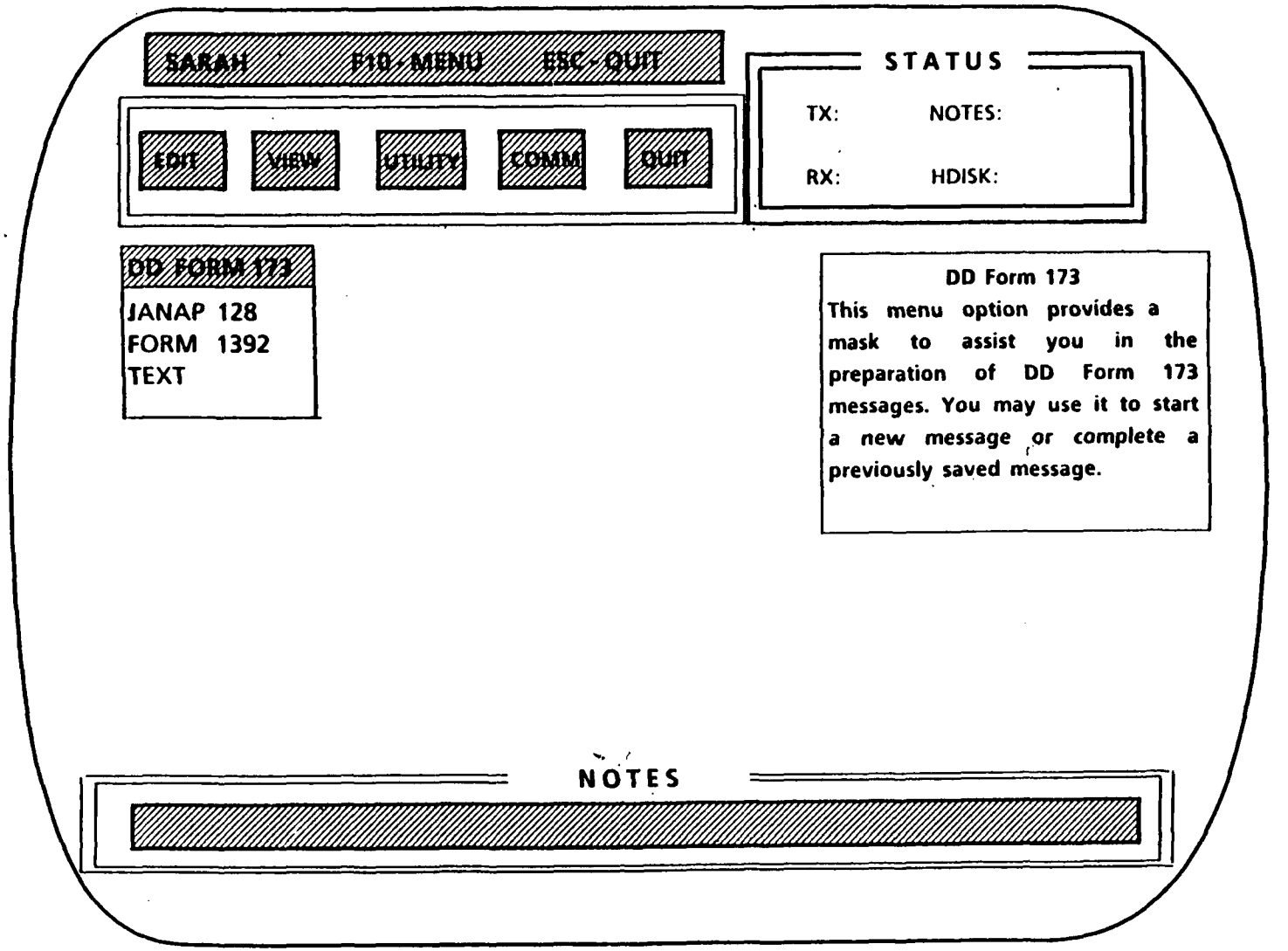


Figure 6-4. **User View.. Help Window**

6.2.2. TOPOGRAPHICAL VIEW

The Topographical View that represented the first level of abstraction in the system design process is shown in Figure 6-5. This graphical representation of the SARAH system is primarily showing:

- The major grouping of functions that make up the system. (Note: major data, or object oriented, groupings did not show up until the next level of abstraction).
- The major interfaces between the groupings of functions. These interfaces can be both control and data.

This picture is a very helpful tool for talking through the thought process that is necessary to design a software system. We found it was quite common to have both this representation and the graphic for the hierarchical view on the screen at the same time.

Note: Since both the project budget and long procurement times did not allow acquisition of automated design tools, we had to make the best use of what tools were available. The group found it very useful to draw the graphical representations on transparent sheets, and use an overhead projector to view them together. The "screen" used was a white, eraseable, marker board. This allowed doing the rough modification on the marker board. When things were a little more permanent, they could then be drawn on the transparency.

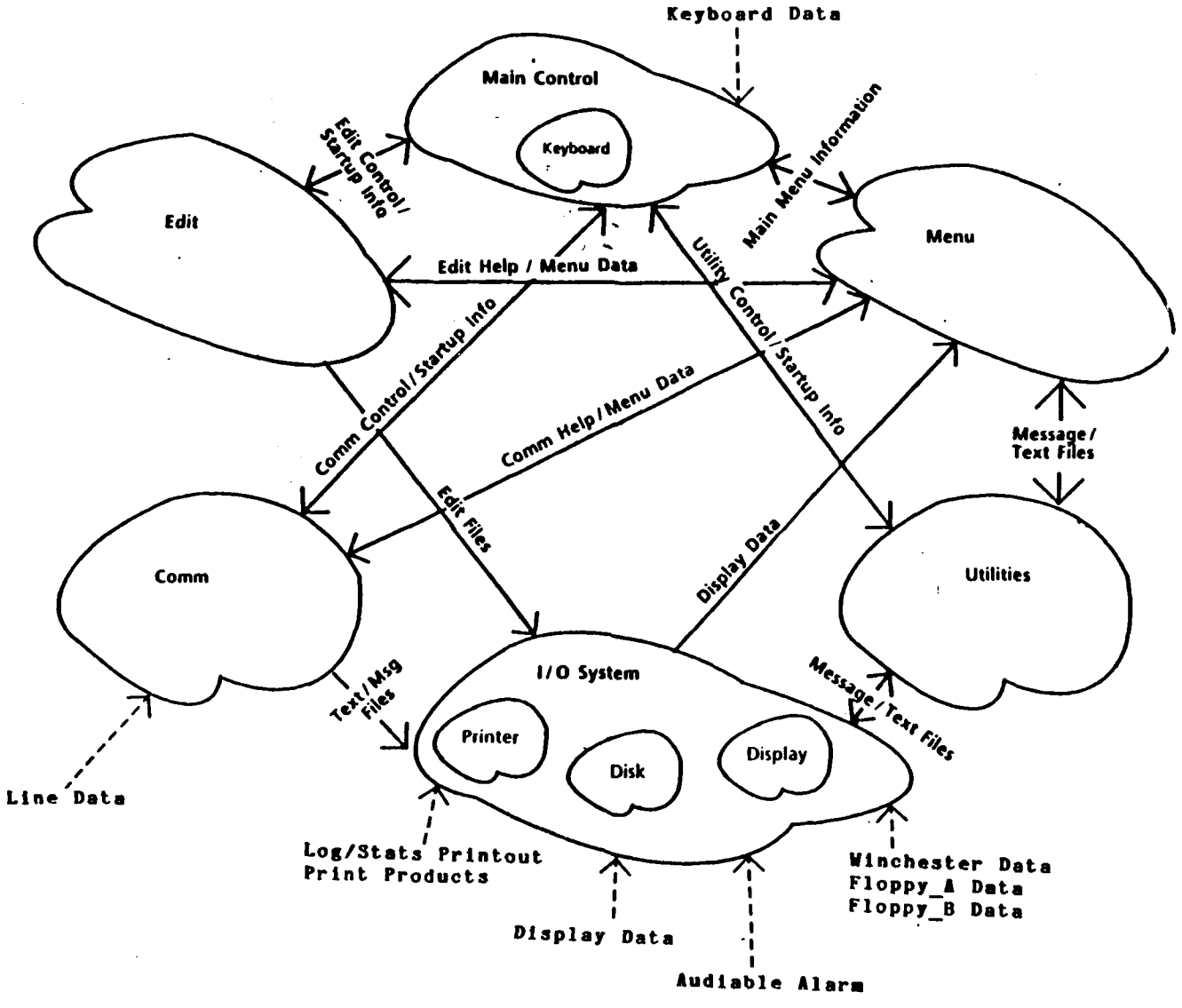


Figure 6-5. SARAH Top Level Topographical Chart

6.2.3. HIERARCHY

One of the two graphical representations that spent the most time being discussed was the hierarchical view. It is shown in Figure 6-6.

The highlights of this graphic are:

- It shows how the system design is to map into the Ada language.
- It allows a forum for discussion of what Ada structures would be most beneficial for data security, code reusability, system reliability, system understandability, and ease of future post development software support. We feel it necessary to discuss these concepts at the highest levels of abstraction. With other languages, the packaging decisions can be made at lower levels of system abstraction. In order to make use of the power of Ada, this process must be done at the very beginning, too.
- It shows compilation dependencies.
- It makes program and data visibility readily apparent. Creating this graphical representation the first time showed us several major problems with how we had visualized the Ada implementation.

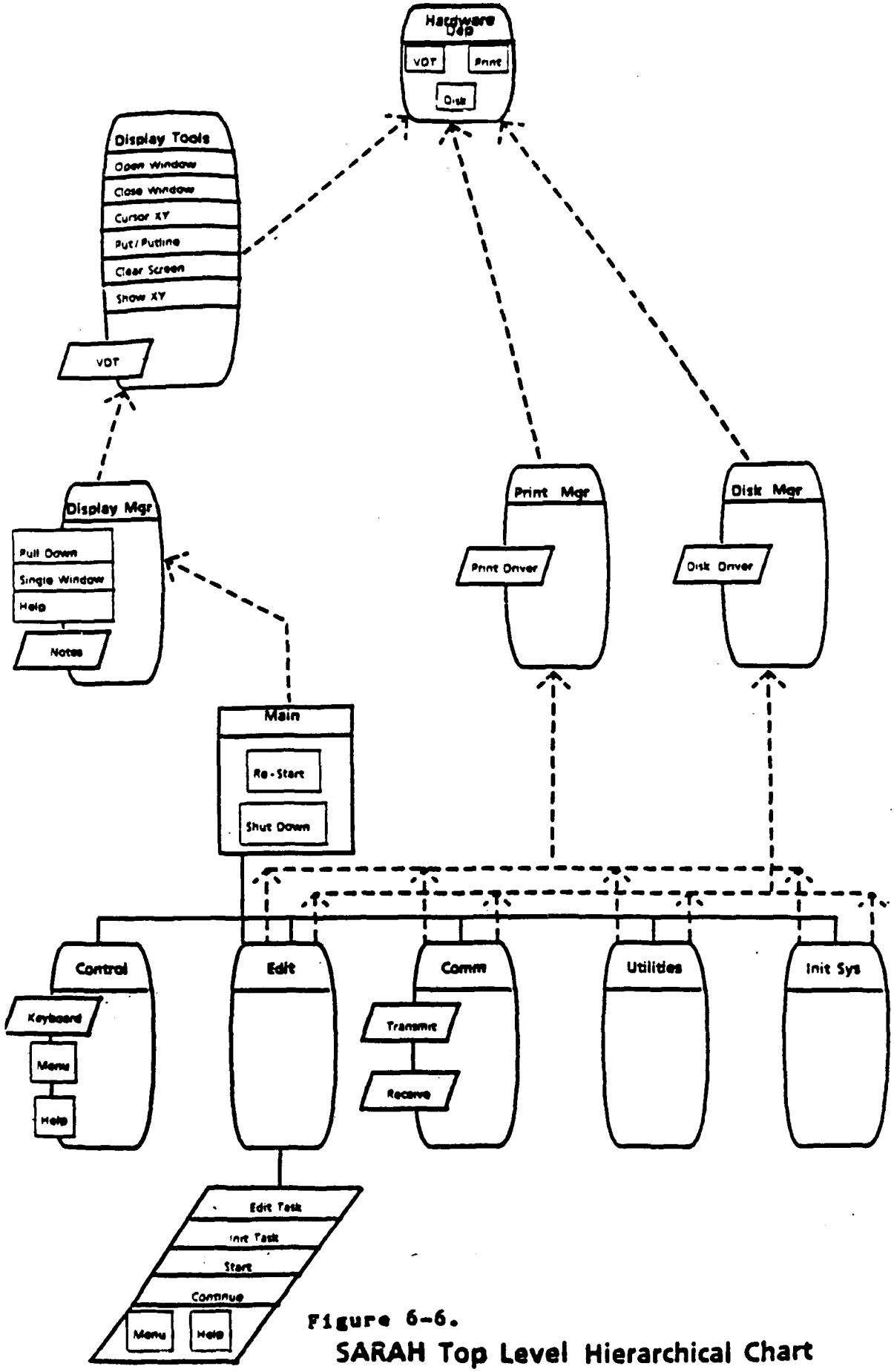


Figure 6-6.
SARAH Top Level Hierarchical Chart

6.2.4. CONCURRENCY

Concurrency should be considered for two reasons:

1) Whenever the system requirements explicitly make it necessary.

2) Anytime an Ada system is being developed. Some functions may be best abstracted using concurrent processes.

The graphical representation for the Concurrency View of the Architectural Model is shown in Figure 6-7. This shows a view of the system concurrency for the highest level of abstraction.

The central vertical line is the concurrency line. Everything connected to it with a horizontal line is considered to be on the line (i.e. concurrent). Those structures that are attached to the concurrency line with a slanted line may become concurrent at times.

The graphical understanding is that the slanted line could fold upward so that it becomes one with the concurrency line. It could also fold back down again to become non-concurrent.

The two functions of transmit and receive are connected to a second vertical line. This representation is to show dependencies. It says "transmit and receive tasks may become concurrent, but only when the communications task is concurrent".

Graphically, we can see that if the slanted line to the communications task folds up to become concurrent, that it would also superimpose the second vertical line on the concurrency line. This would then allow the slanted lines going to "transmit" and "receive" to fold into the concurrency line.

We should point out that the design group had a good bit of trouble agreeing on a graphical representation for concurrency. Additionally, the group had difficulty in determining when a task was concurrent. Was it when the task is created (in the Ada environment, when elaborated)? Was it when it was formally initialized with an explicit rendezvous? Was it when it was available and waiting for work (on the system ready queue)? Or was it when the task was actively doing the job it was designed to do?

Part of the problem is that there seems to be little formal work available on the theories of how to think about concurrency issues. Hopefully, in a few more years, the design texts and methodologies will adequately address concurrency.

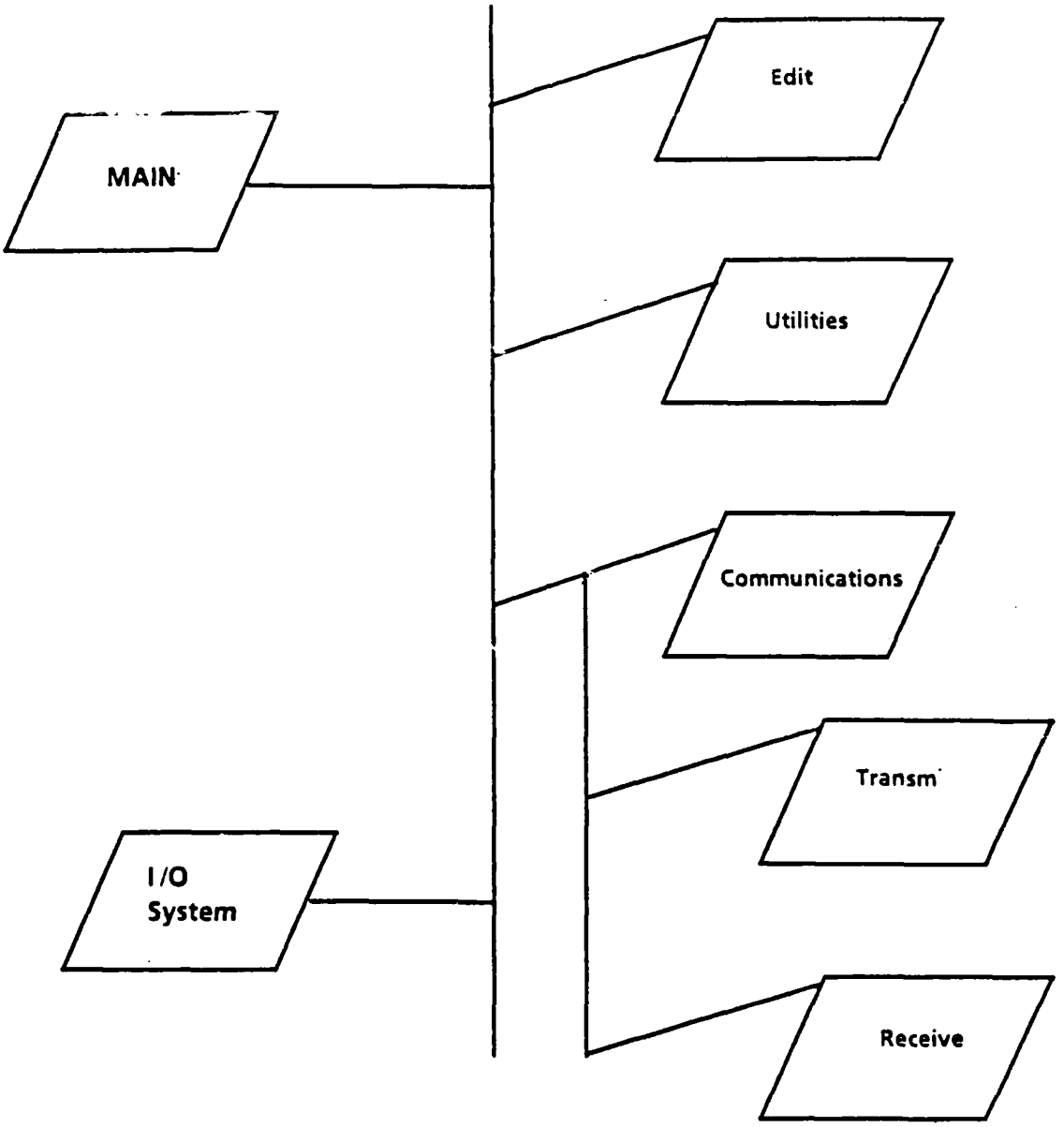


Figure 6-7. **Concurrency Chart**

The best way our design group found to deal with the problem was to look at the idea of logical groupings and dependencies. In Figure 6-7 the I/O System was shown as always concurrent because the entire system was totally dependent on this program unit. If it were not there, nothing could work. The units on the right side of the line are, however, asynchronous in their nature. Any one of them could be completely taken out of the system, and the remainder would still operate.

Thus, at least two units have to be concurrent for anything meaningful to happen (i.e. Main and I/O System). The units labeled Edit, Utilities, and Communications can become concurrent when and if they are needed. The units labeled Transmit and Receive can become concurrent when needed, but only if Communications is already concurrent.

7. BENEFITS OF USING THE SPIRAL AND ARCHITECTURAL MODELS

7.1. ENHANCED COMMUNICATION

In general, graphical representations convey much more information than verbal descriptions. Another fact is that the normal thought processes of good system designers lead them to view a new system using several distinctly different, yet related abstractions.

The Architectural Model of Software Development capitalizes on both of these factors: The model uses four different graphical representations of a system. We believe that these four representations (along with the written narrative that supports the topographical graphic) greatly enhance the ability of the original design team to communicate their understanding of the system to others -- programmers, users, and management.

7.2. MANAGEMENT VISIBILITY

With some approaches to system design, management has to be very confident in the abilities of the designers because the team will not have a great deal of documentation to show until they have progressed well into the design. Most managers feel a bit nervous when the designers say "trust me".

This model provides at least some products that should be very meaningful to management even early in the design process. Management should be very interested in examining the user view documentation so they can know what the product will do in terms they can communicate with the customers of the system.

With the use of the Spiral Model for the lifecycle, and the Architectural Model for the design, it works out very nicely to develop an early prototype to further demonstrate the human interface concepts. Additionally, with Ada as the implementation language, the coding that does the system control for the prototype can, in fact, be the coding for the final system. As analysis, design, and coding progress throughout the Spiral, the "dummy" and "prototype" program units are replaced with "real" program units. This results in far fewer surprises when the "final integration" takes place.

7.3. PROBLEM ANALYSIS AND UNDERSTANDING

At first look, it is natural to think that the process of considering all the different views and documenting them would tend to be confusing. We must admit that the learning curve to become familiar with the objectives of taking each separate view, and learning the graphical notation is not trivial. However, once the basics have been assimilated, we found the overall

design process is faster, more accurate (less backtracking), and far less stressful.

Indeed, this pattern is quite common for most worthwhile things. We often hear the "no pain - no gain" philosophy. For instance, a powerful language like Ada has a very steep learning curve. But once one has made it to the top of the curve, greater productivity, reliability, etc., is possible than with simpler languages.

The human mind is complex, and we probably never come close to fully understanding how it works. However, we feel the Architectural approach to design is closer to the normal thought processes than any of the existing single models. This aproach may be a bit more complex to learn, but if you take the time to learn it, life is made much easier.

7.4. CONCURRENCY SUPPORT

The only fairly popular design methodology we have encountered that explicitly addresses concurrency is Jackson System Design.[8] Because this proprietary model strives to show concurrency and all the other aspects of a system in one graphical representation, we had trouble in understanding the system. Additionally, the fact that it is proprietary makes it more difficult to learn about and to use.

The Architectural Model does address concurrency as a separate abstraction. This feature is going to become more and more important as Ada becomes the language of choice for most projects.

7.5. WORKLOAD DISTRIBUTION

By considering all facets of the system design at the higher abstraction levels, the final shape (i.e. structure or hierarchy) of the system is known early; the final appearance (i.e. user view) is known early. This means that the system components can be divided among different design groups with a high confidence level that everything fits together when it is done.

Additionally, when the Spiral Model and Ada are used, actual coding can start early in the design process. This eliminates the need for bringing on a group of coders late in the design/production lifecycle. Several good coders can be employed during the entire design.

7.6. REUSABILITY AND BOTTOM-UP SUPPORT

In order to take advantage of reusable code it is necessary to do some amount of bottom-up design work. The basic "tools" and "building blocks" must be identified early.

The basic qualities of the hierarchy view and the topographical view allow an object oriented approach to determining tools (i.e. functions, procedures, tasks, etc.) and tool sets (i.e. packages). These tools can then either be acquired from an existing Ada library (if they exist), or provide food for hungry Ada programmers.

7.7. COMPLEXITY MANAGEMENT

As our software systems get more and more complex, techniques to help us manage complexity become of paramount importance. Just as the building architect uses separate pictures to represent different portions of the building design, so the Architectural Model uses different pictures to show different things about the software system design.

The partitioning of the system design into discrete abstractions allows each abstraction to be examined without the mental "clutter" present when using only one system abstraction. Having the separate graphical representation for each abstraction greatly enhances the communication of the system design.

8. CONCLUSIONS AND RECOMMENDATIONS

8.1. CONCLUSIONS

Based upon the work done to design a software system using the Ada language the following conclusions have been drawn:

- The Spiral Model of Software Development is a much superior way of representing the real processes associated with the lifecycle management of software systems.

- The Department of Defense standards for lifecycle management (DoD Std 2167 and associated documents) are more closely tied to the waterfall lifecycle model. Although we have been assured by the people responsible for 2167 that the intent is to allow other lifecycle models, the fact has not been publicized widely. Groups who do not take the time to check with the people responsible may inadvertently be forced into the waterfall model.

- The Architectural Model for Software Design is much superior to any single model or methodology we have found. Its different system abstractions are helpful in working through the design and understanding of the system. They are also very good at communicating that understanding to others.

- The Architectural Model for Software Design is not in itself a methodology; it provides the framework for applying existing software engineering principles. It will hopefully stimulate interest in developing a full lifecycle methodology based on the model.

8.2. RECOMMENDATIONS

Recommendations are:

- The DoD is currently evaluating changes needed to DoD Std 2167 based on lessons learned when implementing Ada. The DoD should seriously consider the Spiral Model of Software Development when they are reviewing the needed changes to DoD Std 2167. They should find it helpful in understanding an overall structure for production and management of DoD software systems.

- Organizations should not blindly standardize on one methodology; rather, methodologies should be chosen to best support specific projects.

- A multi-view, Architectural Model should be used for software design to help manage complexity and aid in

overall understanding.

- More work should be done on models for system design and understanding -- specifically in the area of using multiple abstractions to aid in partitioning complexity, and in the area of providing an integrated system with automated support.

A. REFERENCES

[1] Boehm B.W., "Keeping a lid on Software Costs", Computer World, January 28, 1982.

[2] "Software Technology for Adaptable, reliable Systems (STARS) Program Strategy", Department of Defense, ACM SIGSOFT Software Engineering Notes, April, 1983.

[3] "Report of the DoD Joint Service Tack Force on Software Problems", prepared for the Deputy Under Secretary of Defense for Research and Advanced Technology, July, 1982.

[4] "DoD Digital Data Processing Study - A Ten-Year Forecast", Electronic Industries Association, Government Division, October, 1980.

[5] Boehm B.W., "A Spiral Model of Software Development and Enhancement", TRW Defense Systems Group.

[6] Ward P.T., "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing", IEEE Transactions on Software Engineering, Vol. SE-12 No. 2, February 1986.

[7] Booch G., "Object Oriented Development", IEEE Transactions on Software Engineering, Vol. SE-12 No. 2, February 1986.

[8] Cameron J.R., "An Overview of JSD", IEEE Transactions on Software Engineering, Vol. SE-12 No. 2, February 1986.

[9] Burkhardt B. and Lee M., "Drawing Ada Structure Charts", ACM Ada Letters, Vol VI No. 3.

[10] "Methodman", Ada Joint Program Office, National Technical Information Service (NTIS), accession number AD A123 710.

[11] Cherry G.W., "The PAMELA Designer's Handbook", Thought Tools, Reston Virginia.

[12] Buhr R.J.A., System Design with Ada, Englewood Cliffs, New Jersey:Prentice-Hall, 1984.

[13] "Draft Air Force Information Systems Architecture -- Volume VII ", HQ US Air Force/SITI, 18 February 1985.

[14] Benington, H. D., "Production of Large Computer Programs," Proc. ALnnals of the History of Computing, Oct. 1983, pp. 350-361.

[15] Lehman, M. M., "A Further Model of Coherent Programming Processes," Proceedings, Software Process Workshop, IEEE, Feb.

1984, pp. 27-33.

[16] Parnas, D. L., "Designing Software for Ease of Extension and Contraction," IEEE Trans. S/W Engr., March 1979, pp. 128-137.

[17] D. D. McCracken and M. A. Jackson, "Life Cycle Concept Considered Harmful," Software Engineering Notes, ACM, April 1982, pp. 29-32.

[18] Balser, R., T. E. Cheatham, and C. Green, "Software Technology in the 1990s: Using a New Paradigm," Computer, Nov, 1983, pp39-45.

[19] Department of Defense, Defense System Software Delvelopment, 9 May 1985, Space and Naval Warfare Systems Command.

[20] E. Yourdon and L. L. Constantine, Structured Design, Englewood Cliffs, NJ: Prentice-Hall, 1979.

[21] D. T. Ross and K. E. Schoman, Jr., "Structured analysis for requirements definition," IEEE Trans. Software Eng., col. SE-3, no. 1, pp69-84, Jan. 1977.

[22] "Keeping a Lid on Software Costs", Barrw W. Boehm, Computer World, January 28, 1982.

GENERAL

"Final Report, Joint Logistics Commanders' Workshop on Post Deployment Software Support for Mission-Critical Computer Software", June 1984.

"Analysis and Design For Ada Software", EVB Software Engineering, Inc., 1983.

"Top Level Requirements for Software Engineering Automation for Tactical Embedded Computer Systems (SEATECS)", Naval Ocean Systems Center, August 31, 1982.

"Proceedings of the Air Force Information Systems Architecture Workshop", Air Force/SITI, August 1984.

"Air Force Information Systems Architecture, Vol I -- Overview", Headquarters Air Force/SI, May 1985.