

SANDIA REPORT

SAND85-2348 • UC-32

Unlimited Release

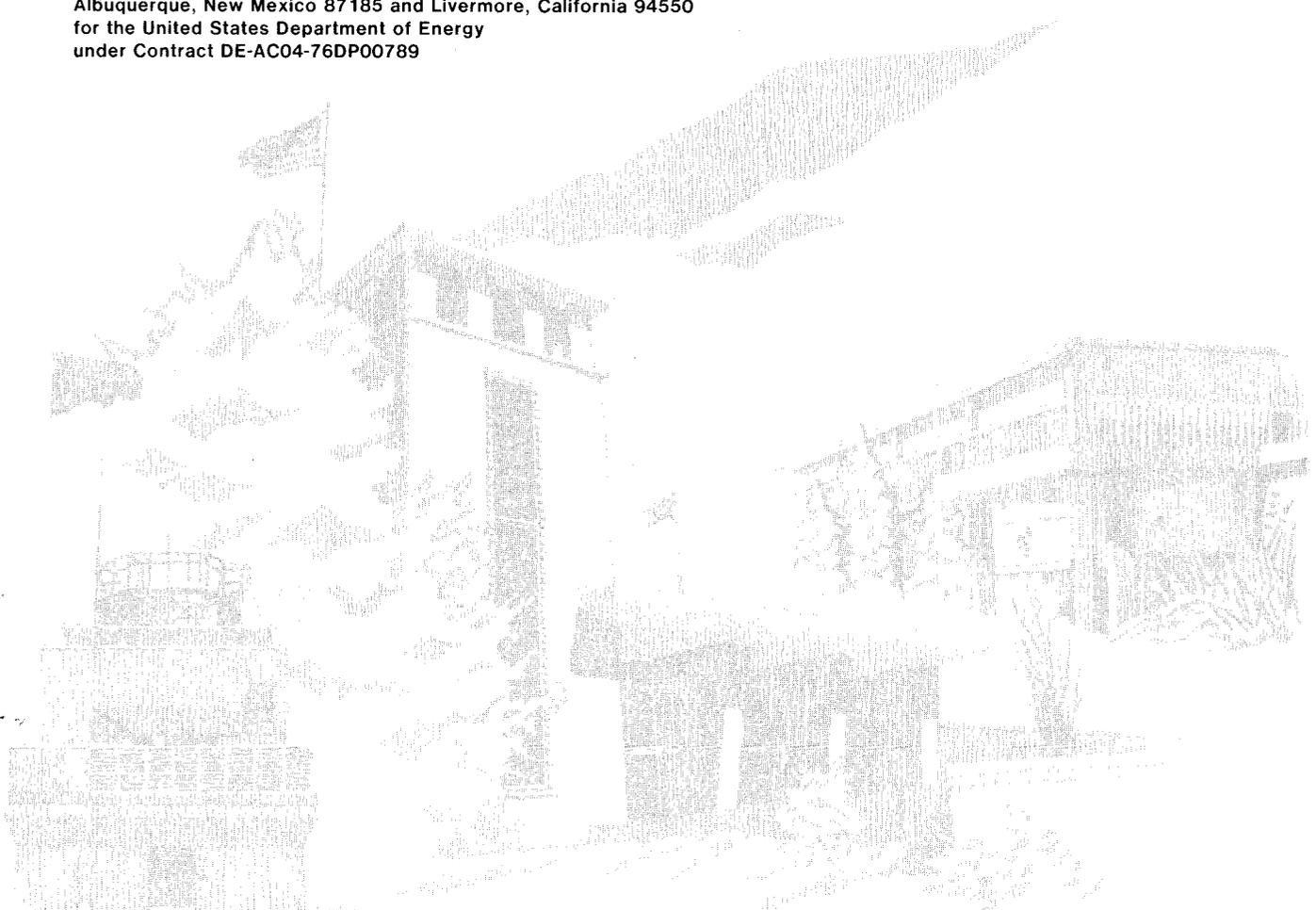
Reprinted September 1992

Sandia Software Guidelines

Volume 5

Tools, Techniques, and Methodologies

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-76DP00789



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A05
Microfiche copy: A01

Distribution
Category UC-32

SAND85-2348
Unlimited Release
Printed July 1989
Reprinted September 1992

Sandia Software Guidelines
Volume 5
Tools, Techniques, and Methodologies

Sandia National Laboratories
Albuquerque, New Mexico 87185

Abstract

This volume is one in a series of *Sandia Software Guidelines* intended for use in producing quality software within Sandia National Laboratories. This volume describes software tools and methodologies available to Sandia personnel for the development of software, and outlines techniques that have proven useful within the Laboratories and elsewhere. References and evaluations by Sandia personnel are included.

Foreword

This volume is one in a series of *Sandia Software Guidelines* intended for use in producing quality software within Sandia National Laboratories. These guidelines, when used in conjunction with the IEEE Standard for Software Quality Assurance Plans, will help ensure that computer programs developed within the Laboratories are usable, reliable, understandable, maintainable, and portable. When complete, the series will consist of the following documents:

- Volume 1: ***Software Quality Planning*** (SAND85-2344)
Presents an overview of procedures designed to ensure software quality. Includes a sample software quality assurance plan for a generic Sandia project.
- Volume 2: ***Documentation*** (SAND85-2345)
Presents a description of documents needed for developing and maintaining software projects. Includes sample document outlines for a generic Sandia software project.
- Volume 3: ***Standards, Practices, and Conventions*** (SAND85-2346)
Presents consensus standards and practices for developing and maintaining quality software at Sandia. Includes recommended deliverables for major phases of the software life cycle.
- Volume 4: ***Configuration Management*** (SAND85-2347)
Presents a methodology for configuration management of Sandia software projects and their associated documentation.
- Volume 5: ***Tools, Techniques, and Methodologies*** (SAND85-2348)
Presents descriptions and a directory of software tools and methodologies available to Sandia personnel.

Acknowledgment

A consensus document like this volume of the Guidelines cannot be produced without the cooperation and hard work of a great many people throughout the Laboratories. The sponsoring Software Quality Assurance Division wishes to thank the members of the balloting group who reviewed and refined Volume 5, and the many contributors of tool evaluations and reviews.

In addition, the editor of Volume 5 wishes to acknowledge and give special thanks to John Franklin, 7254, and Margaret Olson, 2814, for their efforts in preparing, writing, and publishing this volume. The quality of their effort is clear for all to see.

Lynn Ritchie, 7254
Editor, Volume 5

Contents

1 Introduction	1
1.1 Intent	1
1.2 Environment	1
1.3 Applicability	2
1.4 Organization	2
1.5 How to Use This Manual	3
2 Software Methodologies and Techniques	4
2.1 Software Requirements Specifications	5
2.2 Prototyping	6
2.3 Structured Analysis	7
2.4 Information Modelling	9
2.5 Structured Design	10
2.6 Object-Oriented Analysis and Design	13
2.7 Analysis and Design of Real-Time Systems	14
2.8 Software Metrics	16
2.9 Software Inspections	17
2.10 Software Configuration Management	18
2.11 Software Project Management	19
3 Guidelines for the Selection of CASE Tools	21
3.1 Why Use CASE Tools?	21
3.2 Considerations for the Selection of CASE Tools	22
3.3 What Type of Tools Are Needed?	24
3.3.1 Stand-alone Versus Integrated Tools	25
3.3.2 Specific Features of CASE Tools	25
4 CASE Tools In Use at Sandia	32
4.1 Tools for User Interface Prototyping	33
4.2 Tools for Analysis, Design, and Implementation	36
4.3 Tools for Software Configuration Management	51
4.4 Tools for Static Code Analysis	57
4.5 Tools for Reverse Engineering	63
4.6 Tools for Project Management	67
Appendix A: References	70
Appendix B: Glossary and Acronyms	76
Appendix C: Software Tool Description Form	82
Index	85

List of Figures

1	Typical Software Life Cycle	4
2	Sample Data Flow Diagram	8
3	Example Entity-Relationship Diagrams	10
4	Example Structure Chart	11
5	Warnier-Orr Diagram	12
6	Combined Control Flow/Data Flow Diagram	15

Tools, Techniques, and Methodologies

1 Introduction

Software produced at Sandia is important, not only to the function of the laboratories but to the nation's security as well. Increasing reliance on computers has driven the demand for new software that must perform more extensive and increasingly complex functions. These demands for additional function and power require a corresponding increase in the complexity and size of the software needed to satisfy these needs.

The rapid increase in the complexity and size of today's software often overwhelms those who must produce it. The future success of software development depends not so much on larger or faster computers but upon finding and using tools, techniques, and methodologies for the production of software. These aids are needed not only to cope with the magnitude of development efforts but to produce a quality software product.

1.1 Intent

The intent of this manual is to provide a near-current directory of software tools, techniques, and methodologies available to Sandians. The manual is designed to help individuals—designers, managers, quality assurance personnel—involved with computer programs and associated documentation at Sandia National Laboratories to achieve the goal of producing high-quality software. This volume is not intended to teach users how to employ these tools and techniques. Rather, the purpose is to inform users of what tools and techniques are available and how additional information about them can be obtained.

The pace of change in the areas of software tools, techniques, and methodologies has been rapid. To achieve the goal of providing a near-current directory of tools and methodologies will require periodic revisions of this volume. The intent is to update and reissue this volume of the *Sandia Software Guidelines* every few years.

1.2 Environment

Sandia National Laboratories enjoys the uncommon position of fulfilling both research and development, as well as production-oriented missions. Either mission, and certainly the mix of both, lends itself to the creation of "islands of automation." Most software professionals are unaware of what projects are ongoing in neighboring organizations and what software tools have been obtained to support those projects.

Software has played a key role in the Sandia environment for some time. Ample support exists in the form of software tools, techniques, and methodologies for the software developer to enhance product quality. However, it is nearly impossible for any one individual to be aware of all software tools and methodologies available, let alone know what assistance such aids can provide. This volume has been written to assist in providing such information to Sandians.

1.3 Applicability

This manual is designed for use by any organization or project developing or maintaining software, using either Sandia personnel or personnel under contract to the Laboratories. This document provides guidelines to follow regardless of the application of the software, (*e.g.*, WR (War Reserve) or non-WR), the programming language, or the size of the development or maintenance effort.

This manual is a set of *guidelines*, not directives. Inclusion of a description of any tool or methodology in this manual does not imply endorsement by Sandia National Laboratories. Similarly, exclusion of any given tool or methodology should not be interpreted as implied criticism.

1.4 Organization

Volume 5 of the *Sandia Software Guidelines* has been divided into four chapters. Chapter 2 provides an outline of a number of recognized techniques and methodologies used during the software development life cycle. While many of these techniques and methodologies have been automated, it is not the intent of this section to describe the tools that use them. Rather, this section provides a brief description of the underlying method behind each and, where appropriate, references for further investigation on the part of the reader.

The third chapter provides guidelines for the prospective Computer-Aided Software Engineering tool, *CASE tool*, buyer or user. CASE tools often represent a substantial investment, not only in dollars, but in the time required to learn and master them. The guidelines presented here focus on CASE tools in general and on the types of tools which may be used during specific phases of the software development life cycle.

Chapter 4 describes a number of CASE tools currently employed at Sandia. The descriptions of these tools have been provided by Sandians who have used them. These individuals serve as contacts who are willing to answer an occasional question about the tool. Although it can be argued that every software utility constitutes a useful software tool, this chapter does not describe many of the general-purpose programs used at Sandia. Such general-purpose programs include word processors, spreadsheets, editors, compilers, and other utilities.

References for additional information are marked throughout the volume by brackets in this way: [XYZ89]. Details on these references are listed in Appendix A. Appendix B provides a glossary of terms and acronyms used in this volume. For follow-on investigation, the references cited are available through the Sandia library system. Finally, Appendix C provides a form for the reader to submit a description of a tool currently in use at Sandia, but not listed in this volume.

1.5 How to Use This Manual

This manual is designed to be used with the *Sandia Software Guidelines*, references [SSGv1] through [SSGv4], which provide details on suggested software quality practices at Sandia National Laboratories. In addition, references from IEEE, the Institute of Electrical and Electronics Engineers, may prove helpful. The IEEE references are available through Sandia's Design Information Center or from the IEEE Computer Society. The IEEE also provides a single bound publication of all their current software engineering standards [IEE89].

2 Software Methodologies and Techniques

This chapter describes a number of methodologies and techniques used in the development of software. The intent is not to provide a comprehensive explanation of each method; such an attempt would fill many volumes. Rather, the aim is twofold. First, each method will be briefly described and its use placed in the context of the software development life cycle. Many of the techniques and methodologies described here have been embodied in CASE tools—computer programs used for the development, testing, or maintenance of software. The second aim is to highlight those features of a software development method that are relevant to describing the CASE tools listed in Chapter 4 of this volume. Where appropriate, references have been provided for further investigation.

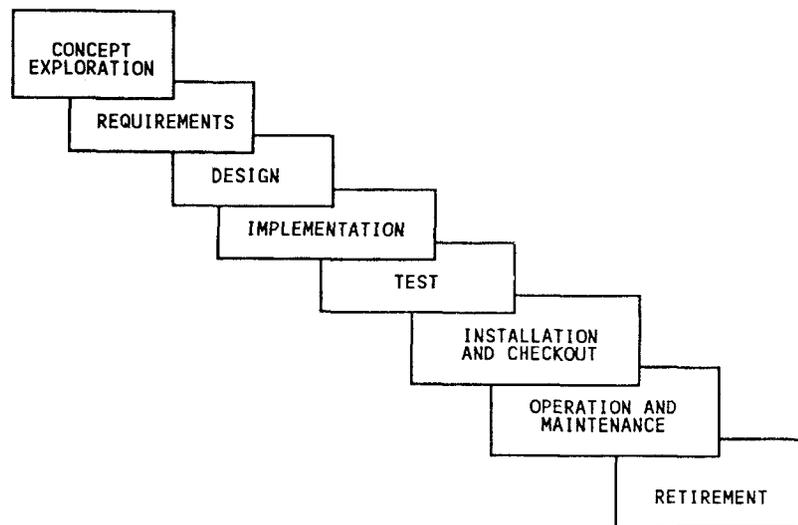


Figure 1: Typical Software Life Cycle

A model representing the different phases of software development and use is shown in Figure 1. This so-called “waterfall” model is one of the popular, accepted ways of representing the development and use of software. While the process of developing software is not always as neat or compartmentalized as the model implies, it does serve a useful purpose and will be referenced to describe the context in which different software techniques and methodologies are employed. Some of the activities and methods discussed are restricted to a particular phase, while others apply throughout the entire life cycle. The interested reader is referred to the third volume of these Guidelines [SSGv3] for a detailed description of the life cycle, along with the activities and deliverables recommended for each phase.

Before describing any particular software methodology or technique, it would be prudent first to discuss and define some basic terms and concepts. Among the more important of these is *software engineering*. Software engineering is the systematic approach to the specification, development, testing, operation, maintenance, and retirement of software. There are a number of techniques and methodologies that together form the core of the discipline known as software engineering. These methods and techniques enhance the quality and productivity of the activities related to software development. For the purposes of this volume, the subject of software engineering can be divided into two subject areas. One of these divisions includes those topics that concern software *product* technology. Compilers, languages, and parallel architectures can be considered examples of software product technology. The remaining division includes the subjects of concern to this volume—those having to do with a *process* technology. Software process technology as it relates to software engineering is concerned with the processes of developing software and managing the development of software. It is the process technology aspect of software engineering that will be examined in this chapter. That software engineering is a relatively young and still developing discipline should be understood.

No attempt will be made to formally distinguish between a software methodology and a technique. When used in conjunction with software development, the difference is often quite blurred. DeMarco [DEM87] provides a definition of *methodology* adequate for most purposes: “A general systems theory of how a whole class of thought-intensive work ought to be conducted.” A *technique*, on the other hand, may be regarded as less encompassing or comprehensive than a methodology. From one perspective, one might think of a methodology being comprised of one or more techniques, together with concept or theory that makes it cohesive.

The discussion of specific techniques and methodologies is presented in an order that roughly follows the chronological ordering of activities as they occur in the software life cycle.

2.1 Software Requirements Specifications

Of all software life cycle phases, it is generally accepted that the requirements phase is the most important. The aim of activities conducted during this phase is to reach an understanding of what functional and performance attributes are required from the software in question. Achieving this understanding is not trivial. The consequences of proceeding into software development with requirements poorly defined or misunderstood can be great. The need for requirements to be defined and documented in a clear and unambiguous fashion so that all concerned parties can review and agree to them cannot be overemphasized. To facilitate review, what is needed is a standard method of specifying requirements, a uniform format for guidance when producing software requirements documentation. Several such formats exist.

The usual Department of Defense or military contract with Sandia requires the generation of software documentation, including requirements. This documentation is to be prepared in accordance with the Defense Department standard DoD-STD-2167A, *Defense System Software Development* [DOD67]. Such documentation, known in DoD parlance as *Data Item Descriptions*, or DIDs, must be written to a rigid format that goes so far as to specify the labelling and contents of each and every section in detail.

A less rigid approach to developing requirements documentation is recommended in the *IEEE Guide to Software Requirements Specifications* [IEE84a]. This standard presents a general outline for the documentation of software requirements. Included are guidelines and recommendations for writing a good Software Requirements Specification together with some of the common pitfalls to be avoided. The IEEE format is the one suggested by Volume 3 of these Guidelines [SSGv3].

A standard outline and format for specifying requirements contributes to common comprehension of those requirements. However, one should not take away from this discussion that the only way to document requirements is with a text-based specification. Textual documentation has its particular strengths and weaknesses. For especially large or complex projects, text-only documentation may sometimes hinder understanding, concealing even the simplest functional requirement under a blanket of words.

Several other useful approaches to developing and documenting requirements are available. Some of these methods are graphical in nature, using pictorial representations to specify functionality or other requirements. The use of pictorial representations clearly illustrates the adage that one picture is worth a thousand words. One of the best known techniques, *structured analysis*, is described in Section 2.3.

While other means of specification do have certain advantages over a text-based specification, one should not use them to the exclusion of a textual requirement. Rather, these other methods should be used to complement the textual specification.

2.2 Prototyping

The importance of documenting requirements was stressed in the previous section. There are times, however, when little possibility exists to define requirements adequately. At other times functional requirements may be known, but the way the computer program interacts with users may need to be refined. Under such circumstances, the technique of *prototyping* can be used. Using a prototype permits one to learn more about the problem at hand, trying out new algorithms or verifying capabilities.

Prototyping, when applied to software, involves the development of a program to emulate one or more functions yet to be clearly defined. A software prototype is a minimally functional system used to prove the feasibility of a concept or demonstrate some aspect of its functionality to a potential user. The prototype may be developed in the same manner as any other program. More commonly, specialized tools for the creation of prototypes are being used. *Rapid prototyping* using these specialized tools permits one to quickly and economically produce many versions of a prototype within a relatively short period of time.

User interface prototyping is a special case of rapid prototyping that focuses on the definition of the human-computer interface. This interface is both visual and behavioral. Guidelines for the design and evaluation of user interfaces are given in [CHA86]. Prototyping the interface allows both the software developer and the user to consider together how information is displayed on the screens, and the manner and order in which responses are handled. The function and appearance of the program are simulated by using only a mock-up, or facade, to imitate a fully functional program.

The prototyping approach has several benefits and pitfalls. With little investment, prototyping a part of the system (*e.g.*, user interfaces) may be sufficient to define an appropriate approach to specifying requirements for the rest of the system. In parallel hardware-software development efforts common at Sandia, prototyping may be the only way of visualizing the final product before the availability of the host hardware. A common pitfall to be avoided is allowing the code for the prototype to become the basis for the final product. The prototype, suited for exploring concepts and ideas, may well be poorly designed, unstructured, and not easily maintained when incorporated into the deliverable software.

2.3 Structured Analysis

Of all of the methods available for developing functional requirements for a software system, DeMarco's [DEM79] Structured Analysis methodology is the best known. This method is the underlying basis of many CASE tools, and has been enhanced by others to encompass more specialized forms of specification [HAT87][WAR85]. A variant of DeMarco's approach has been popularized by Gane and Sarson [GAN79].

Structured Analysis is essentially a set of techniques for modelling the functionality of a system. The technique is not used to design software, but to model the processes the software is to eventually implement. The product that results is a structured specification. Interestingly, the method has been used successfully to model a wide variety of processes, not just those performed by software.

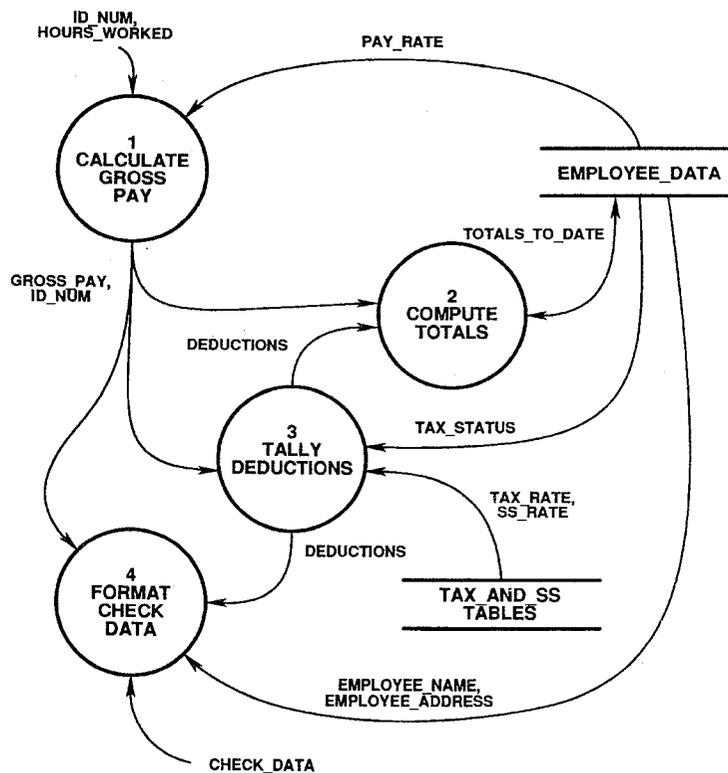


Figure 2: Sample Data Flow Diagram

At the heart of this method are pictorial representations called *data flow diagrams*. An example data flow diagram is shown in Figure 2. The major elements of the diagram consist of data flows, illustrated by the labelled arcs, and *processes*, represented by circles. Other symbols denote sources and sinks of data external to the system, as well as the internal storage of data. The diagram illustrates in a simple and elegant manner the logical flow of information in a system, together with the processes that perform transformations on the information. Each process may be further decomposed into additional processes, creating a structured, hierarchical decomposition of the overall specification. Complementing the use of data flow diagrams are two other techniques, the process specification and the data dictionary.

Process specifications, known also as *p-specs* or *mini-specs*, provide a textual description of *primitive* processes in the hierarchy of diagrams (*i.e.*, those processes that have not been subdivided and described by still more processes). Process specifications can be represented in a number of ways: “structured” English descriptions, truth tables, or decision trees.

The *data dictionary* contains a description of information represented by the flows in the diagrams. Individual data elements in the dictionary are usually defined using a rigid and formal syntax. One of the more popular and effective means of describing this syntax is to use a *metalanguage* such as the Backus-Naur form (BNF) [HEH84].

As an analysis tool, DeMarco's method possesses several shortcomings. While the method does a good job of capturing functional requirements, it does not allow other types of important requirements to be defined. For instance, no provisions are made for the specification of performance limits, which are better described in a textual format. The state dependent behavior and timing aspects common to real-time systems remain undefined in conventional data flow diagrams, as do the important relational attributes between data elements for data base applications. DeMarco's basic technique has been enhanced to include real-time system requirements as discussed in Section 2.7.

2.4 Information Modelling

For a large class of software systems, the associations between pieces of information are just as important as how that information is processed. As an example, a data base system might typically perform little processing of the information it is responsible for storing. Instead, the system will be used to retrieve information selectively based on specified relationships or associations between elements of the data base. A traditional structured analysis performed for such a system may be deceptively simple because of the small number of data processing requirements. Yet other factors are involved in understanding and specifying such a system.

Information modelling is a technique used to analyze and model the associations and relationships between data. The technique is an important method for understanding and developing requirements for relational data bases, expert systems, and other types of systems where data associations and relationships are of primary importance. Information modelling is sometimes referred to as *knowledge engineering* or *knowledge analysis*.

Chen [CHE77] originated what is perhaps the best-known information modelling technique, the *entity-relationship* approach. The method relies on diagrams to illustrate relationships between things of importance, or *entities*. These diagrams are referred to as *entity-relationship diagrams*. Figure 3 illustrates several different relationships modelled using this technique. Of note is the way in which relationships are represented succinctly and unambiguously. For example, from the relationship depicted in center diagram of Figure 3 one can determine that:

- a child is the offspring of one and only one mother
- a mother may be the parent of one or more children

The entity-relationship approach is not the only technique used for information modelling. The *Nijssen Information Analysis Method*, or NIAM, [NIJ87] is another method currently popular in Europe and was recently introduced into the United States. The NIAM technique relies on the use of graphic illustrations of associations between entities and superficially resembles Chen's entity-relationship approach.

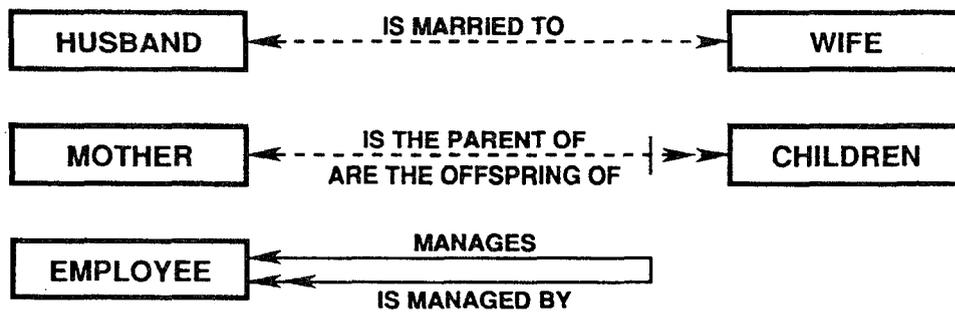


Figure 3: Example Entity-Relationship Diagrams

Yet another, and more recent, approach to information modelling is based on object-oriented concepts. Shlaer and Mellor [SHL88] conceived of their technique to be used for the analysis of systems in which data is of primary importance. Although sometimes referred to as a method for object-oriented systems analysis, it is in reality a specialized form of information modelling. Object-oriented system concepts are discussed in detail in Section 2.6 of this chapter.

2.5 Structured Design

During the requirements specification phase, one is concerned primarily with obtaining an understanding of the problem to be solved, and transforming it into a written specification. During the design phase, one seeks specific solutions to the problem in the form of a plan for the software system to be implemented. This plan specifies how data is to be organized (*i.e.*, data structures), what individual components (*e.g.*, procedures, subroutines) are to comprise a system, and the way in which those components are to interface and share data. The design describing a software system is analogous to an architect's blueprint: a detailed architectural plan for construction.

Structured design methods are a disciplined approach based on specific rules and principles. In contrast to structured analysis, there exists no single, unified approach to structured design. Rather, various methods have been espoused by their proponents. Fundamentally, the various methods vary little. All have as their goal to produce a design that is *modular*. A modular design is one that minimizes the *coupling* between modules, and maximizes the *cohesion* within each module. Modularity enhances maintainability and greatly facilitates a structured approach to testing.

Structured design methods also share another important characteristic. From a methodological standpoint, the design of a system is derived from a hierarchical decomposition of functional specifications. Modules in the resulting hierarchy are structured according to the functions and procedures they are to implement. Structured techniques can be considered *procedure-oriented*, in contrast with *object-oriented* methods discussed in Section 2.6.

Perhaps the most popular structured design method is attributed to Yourdon and Constantine [YOU79], and has been emulated and enhanced by others, notably Page-Jones [PAG80]. This method focuses on the data flow diagrams from structured analysis as the starting point for a design. The overall design of the system is documented using *structure charts*. The structure chart is similar in appearance to an organizational chart (or the familiar *HIPO* charts), but has been modified to show additional detail. Structure charts can be used to display several types of information, but are used most commonly to diagram either data structures or code structures. An example structure chart is shown in Figure 4. Figure 4 illustrates the hierarchy of modules in the design, which modules invoke other modules, and the data and controls that are passed between modules. The structure chart would be accompanied by a *module specification* that describes the logic and function of each module.

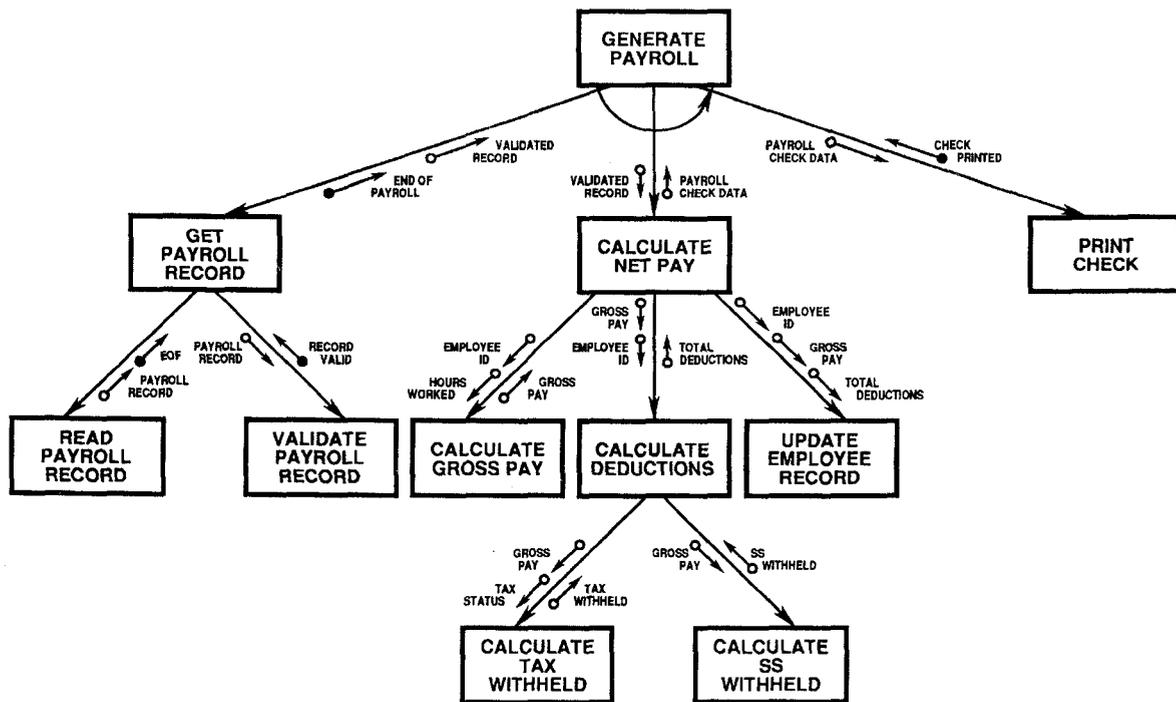


Figure 4: Example Structure Chart

CASE tools implement the structured design methods of Yourdon and Constantine almost to the exclusion of other methods. However, many other popular design methods exist. One of these design methods is the Jackson Structured Design (JSD) technique, developed by Michael Jackson [JAC83]. Jackson's method is a comprehensive six-step method that integrates both software analysis and design. A quite different method of design is the Warnier approach [WAR78], popularized in the United States by Ken Orr. Orr adopted many of Warnier's concepts of program design, and embedded them in a graphic technique known as Warnier-Orr diagrams [HIG77], [HIG78]. An example Warnier-Orr diagram is shown in Figure 5. These methods are by no means the only structured design techniques available. Citations for other structured design techniques are provided in the references for this chapter.

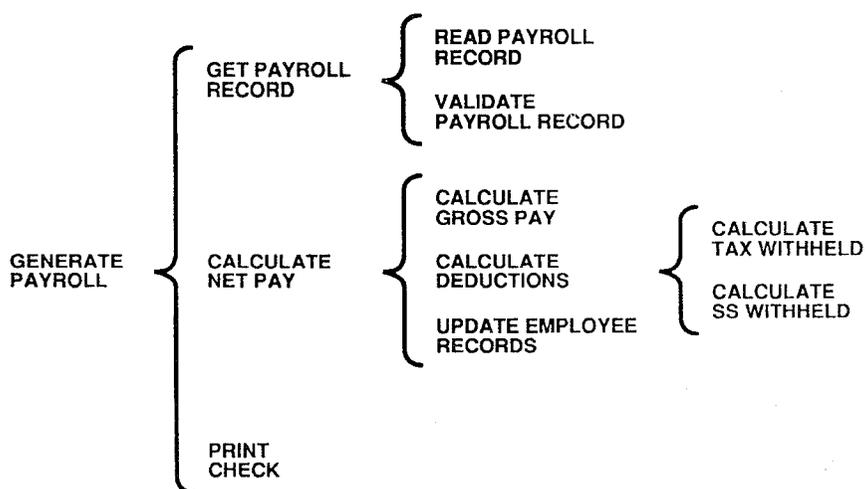


Figure 5: Warnier-Orr Diagram

Before ending this discourse on structured methods, it is appropriate to discuss what is meant by structured programming. Structured programming has its roots in the earliest days of software engineering, and came about from research in attempting to develop provably-correct programs. Dijkstra [DIJ73] among others has written much of the seminal work in this field. Structured programming languages and techniques generally restrict the programmer to a well-defined set of logic constructs for coding. These logic constructs are used for operations like iteration (DO — WHILE or FOR — NEXT) and decision-branching (IF — ELSE IF — ELSE). Specifically uncharacteristic of structured programming is unconditional branching (*e.g.*, using GOTO). Use of the approved logic constructs, together with other practices associated with structured programming, increases the readability of the code and simplifies its logical complexity. Kernighan and Plauger [KER78] are an excellent source of advice on programming style, as is the third volume of these Guidelines [SSGv3].

2.6 Object-Oriented Analysis and Design

In recent years there has been an increased focus on an approach to the analysis and design of software called the *object-oriented* approach. Object-oriented approaches offer an alternative to the traditional “structured” methods of analysis and design in software development. The concepts and techniques of object-oriented analysis and design have their origins deeply rooted in object-oriented programming.

Systems analyzed using conventional structured techniques are modelled as collections of processes or procedures. Each procedure receives data, performs some transform or calculation using the data, and produces new data as a result. While the *process* is the central entity used by structured methods, the *object* is the focus of object-oriented techniques. Objects are abstractions of real-world items, and are not to be confused with the actual items themselves. Objects are comprised of data plus the operations which act on that data. Each object has associated with it one or more defined operations. In software systems, objects are represented by data structures and accessed through program *messages*. Objects communicate with each other through message passing. Information passed to objects does not constitute data in the “structured” sense, but rather messages informing the object what operation it is to perform on itself. These operations, or *methods*, are hidden within the structure implementing each object. In object-oriented systems, objects are typically grouped into *classes*. A class can be subdivided into *subclasses* which share common features of the original class.

There are distinct advantages in the use of object-oriented techniques. Since system design depends on objects which realistically represent the real-world objects they model, the understanding of both the design process and the system model is improved. Enhanced modularity provides a clear advantage in both the development and maintenance processes. Increased modularity together with the properties derived from objects and classes yield another benefit—the potential for developing truly reusable software. An often-mentioned benefit is that an object-oriented approach may be the only viable one for the development of extremely large software systems. As systems become larger, structured methods of analysis and design add additional layers of functional complexity vertically, making them more difficult to understand and implement. In contrast, a system is enlarged when using object-oriented methods by the addition of more objects. No additional levels of complexity are required to describe each object.

In general, an implementation using object-oriented programming languages may, at first, seem “very different” and perhaps more difficult when compared to procedure-oriented languages. It can be argued that the implementation is in fact easier because of the nature of the object-oriented design. The nature of an object-oriented language is such that the programmer develops and uses class libraries of code extensively.

The few analysis methods available for object-oriented systems are based on information modelling techniques similar to those discussed in Section 2.4 of this volume. One such analysis method has recently been developed by Shlaer and Mellor [SHL88]. Specifics of design techniques used for object-oriented systems often depend on the particular language used for the implementation. At the present time, there exists no uniform method of object-oriented software development. Brief and succinct overviews of object-oriented design are given in [PAS86] and [TES86], while more specific, in-depth treatments can be found in [COX86] and [MEY88].

While any programming language could be used for development of object-oriented systems, in practice the language should facilitate the use of the key constructs and attributes of object-oriented programming. There exist three classes of languages suitable for object-oriented programming. First are those normally associated with procedure-oriented languages that possess a rich set of features with which object-oriented systems can be implemented. An example of this class is the Ada programming language. Another class of languages are supersets of more conventional languages, such as C, Pascal, and Forth, that have been enhanced. The final class of languages includes those that have been specifically developed for object-oriented programming. Of these, the Smalltalk programming language is probably the best known.

2.7 Analysis and Design of Real-Time Systems

The term *real-time* describes a wide variety of systems which cannot be neatly encompassed by a short definition. However, definite traits do characterize real-time systems. Real-time systems receive input data directly from the environment they interact with and often utilize special hardware designs or architectures optimized for a particular application. Systems required to respond interactively with users, or which are responsible for the control of physical processes, are generally portrayed as being real-time systems. *Embedded* systems are typically real-time systems. Secondly, these systems are required to provide a response which is keyed to the time scale of the process being executed. A rapid response alone is not a criterion; a real-time response may vary from nanoseconds to hours, or even days. The interested reader is referred to [MEL83] for a more complete overview of real-time systems.

The structured techniques for analysis and design of software discussed earlier in this chapter work well for the majority of data processing systems. However, real-time software systems possess other characteristics that make them different from typical data processing systems. For example, a system responsible for air traffic control is quite different in nature from one that performs payroll calculations. The air traffic control system must perform a number of different tasks for which timing and synchronization are at least as important as the processing of data. These timing and synchronization considerations are not addressed by conventional, "structured" approaches to software analysis and design.

From a software engineering standpoint, the specification and design of real-time software is difficult, particularly for large systems. The difficulty arises from the additional degree of complexity involved. Not only must the software specialist consider the processing of data, but the timing and sequencing of the processing. In order to approach the problem in a systematic fashion, techniques and methodologies must be designed specifically to accommodate real-time systems. Analysis and specification techniques must describe the modes of behavior, or *states*, of a system together with the conditions that cause the system to change from one state to another.

A number of analysis methods have been developed to handle real-time systems. In general, they are either too simple, complex, or specialized to use for a general class of problems. Perhaps the best known, and most often implemented by CASE tools, are those of Hatley and Pirbhai [HAT87], and Ward and Mellor [WAR87]. Both methods essentially employ DeMarco's structured analysis approach discussed in Section 2.3, but with added extensions to accommodate the specification of real-time systems. These extensions provide graphical representations to model control, timing, and synchronization. Analogous to the data flow diagrams, data stores, and process specifications in the conventional method are control flow diagrams, control stores, and control specifications. Control flows are denoted similar to data flows, using dashed rather than solid arcs. Figure 6 illustrates the Ward-Mellor approach to structured analysis in which the data flow diagram and control flow diagram are combined into one diagram.

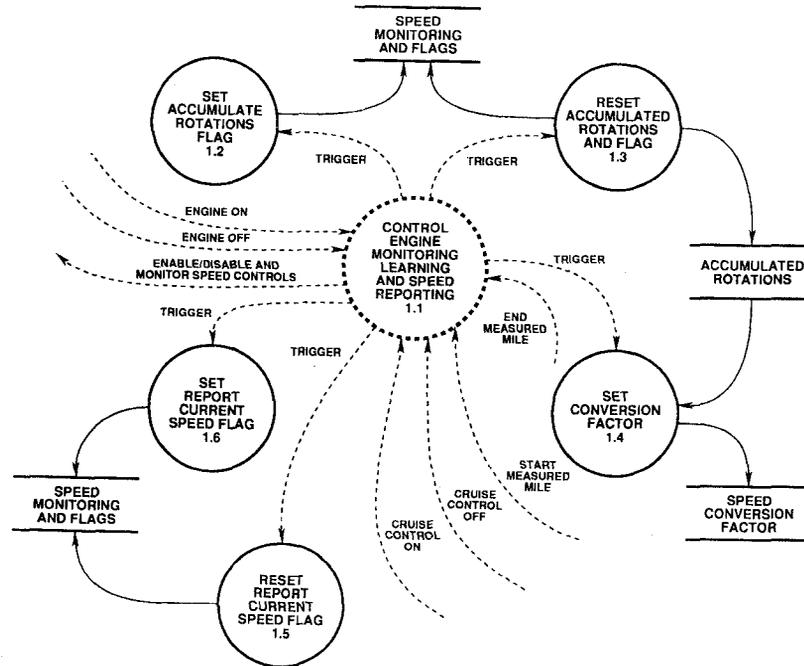


Figure 6: Combined Control Flow/Data Flow Diagram

The design and implementation of real-time systems generally follow the methods already described for structured design and programming. Generalization is difficult, however, since the design aspect of a real-time software system is often constrained by the pre-existing hardware and system support available. For example, a particular external event could be polled, or handled as an interrupt. However, the decision many times is determined by a pre-existing hardware design. Data structures and module interfaces may have to be designed to be compatible with existing system software designed to handle real-time functions related to timing and device input/output.

2.8 Software Metrics

The statement is often made that one cannot control that which cannot be measured. Almost without exception, all engineering and scientific disciplines make use of quantitative measurements to judge the “goodness” of a process, product, or result; such measurements often form the entire foundation of that discipline. Can particular attributes of software be subjected to similar measurements? If not, how can one make prudent engineering judgments about software quality, the resources that should be allocated to a development effort, or even where the development process needs refinement? Some insist there must be a better way than the traditional “wet finger in the air” approach. Software metrics attempt to address these issues.

A *software metric* is a quantitative measure of some attribute of either software or the process used to develop software. Note that there are many measures which quantify some aspect of software or computer behavior: MIPS, FLOPS, Kbytes, and so on. The discussion of such quantitative measures in this section is necessarily restricted to those that deal directly with software or the production of software. Some measurements are used to provide an indication of how complex the software is or how much difficulty one might anticipate in its maintenance. Others are used to quantify aspects of software quality and provide a relative degree of confidence in the reliability of the software product. A third area addressed by software metrics is project management: cost estimation and prediction.

Complexity metrics were historically among the first attempts to quantify a feature of software. Complexity metrics are derived from the logical complexity of code by, for example, counting decision points or number of variables. Using complexity metrics, one can infer the relative degree of effort required to test or maintain a piece of code. Two of the better known and used complexity metrics were developed by McCabe [MCC76] and Halstead [CHR81]. Both metrics derive their measurements directly from source code statements.

Measurements of software reliability and similar “quality” attributes cannot be determined solely from an examination of the source code. Software quality metrics rely heavily on data obtained during the development and maintenance phases for the software, and demand that careful records be kept from inspections, testing, and field reports. Both Gilb [GIL77] and Boehm [BOE78] reported much of the initial work in the area of software reliability metrics. A survey of recent work in this area is present by Shefif *et al.* [SHE85].

Perhaps the most practical use of metrics is for estimating development effort and projecting completion dates. Many other engineering disciplines have reduced the estimation and projection procedures to look-up tables and simple formulas. DeMarco [DEM82] convincingly argues that stable software development environments implementing structured methodologies can successfully use similar procedures. Other notable work in this field has been conducted by Boehm [BOE81], who has developed several sophisticated models used for estimation. Like many other software metrics, the ability to accurately estimate and project relies on the existence of a base of historical data characterizing past efforts.

The conceptual and theoretical underpinnings of software metrics have been written about for over a decade. However, some difficult issues must be addressed when collecting or interpreting metrics. Many of these problems result from the fact that some metrics are not a measure of some physical phenomena but the reflection of an intellectual process. Due to the relative immaturity of the science of software, many of the elements to be measured are themselves poorly defined and not well understood. For instance, one unresolved question is whether the traditional concept of reliability measured in terms of mean-time-to-failure has any real meaning when applied to software. Estimates and cost projections are typically calculated based upon data collected during previous development efforts. Are the estimates produced of any real use if they are based on a software development process that is largely unstructured and chaotic? Those who wish to pursue or further investigate these issues may wish to refer to the work of Jones [JON86], and Grady and Caswell [GRA87].

2.9 Software Inspections

Although the waterfall model in Figure 1 portrays phases of software development, it does not show any quality assurance or control activities other than testing. Encapsulated within each of the development and maintenance phases shown in the waterfall are the activities complementing development to enhance the software quality. These activities are varied, but all have in common the purposes of reviewing the progress made thus far, ensuring that the activities performed are in harmony with those of the previous phase (*verification*), and ensuring that the current development effort is in line with the specifications of the original software requirements (*validation*).

Methods of reviewing activities range from formal presentations to casual peer reviews and walkthroughs. These methods vary greatly as to their efficacy in improving the quality of the software product. In 1972, IBM created a formalized review process that has since been shown to significantly decrease the number of software defects while correspondingly increasing programmer productivity. Generally credited to Michael Fagan [FAG76, FAG86], *software inspections* are formal peer reviews in which a software product is evaluated solely to detect faults, violations of standards, and other problems. Correction of the defects detected takes place outside of the inspection process. Originally used on software designs and code, the inspection technique can be used for evaluating requirement specifications, test plans, and other software documentation. The inspection technique has been formally incorporated into software development by firms large and small (*e.g.*, IBM, Hewlett-Packard, AT&T), and many conduct classes in its use.

The success of inspections is due to a formalized, structured approach that enhances the synergy of the inspection team. Ground rules for the conduct of the inspection are specified, as well the role each person plays during the inspection. Emphasis is placed on preparation before the meeting; generally, as much time as will be spent in the inspection. The amount of material to be reviewed is restricted to that which can be comfortably covered in about two hours. A record of defects discovered is kept for the product's author to address. The underlying concept for the inspection is not to evaluate the author or consider alternative designs but to improve the software product by helping the author identify defects.

Managing inspections does not require a knowledge of software engineering concepts as much as an understanding of human nature and group psychology. Formal training in the inspection technique is desirable, particularly for team leaders. The interested reader is referred to both Fagan [FAG76] and Yourdon [YOU85] for additional information on software inspections. Volume 3 of these Guidelines also contains a description of software inspections, as well as blank copies of report forms used throughout the inspection process.

2.10 Software Configuration Management

Software frequently undergoes many changes during its lifetime. This volatility demands that particular attention be paid to controlling and tracking changes made during the development and maintenance of software products. No specific software engineering methodology or discipline addresses this issue. Instead, these issues are properly viewed from a management perspective.

The term *software configuration management* describes the set of activities used to control and track changes to software. Specifically, it encompasses activities that

- identify and define software and hardware items of concern (*configuration items*)
- control the release and change of configuration items throughout their lifetime
- record and report the status of configuration items, together with requests for changes to them
- verify the completeness and correctness of configuration items

Configuration Management [SSGv4], the fourth volume of the *Sandia Software Guidelines*, provides an in-depth discussion of software configuration management, as well as associated guidelines and recommendations specific to the Sandia software environment. This volume addresses software configuration management activities during development to assure consistency among configuration items, and the use of the Sandia drawing definition system to address configuration management concerns after system release. A brief discussion of configuration management can also be found in Volume 3 of these Guidelines.

Another useful guide is provided by the IEEE in the form of a standard for developing Software Configuration Management Plans [IEE83a]. This standard provides an outline for, and the recommended contents of, a configuration management plan. The standard is equally applicable to both individual software projects, and the development activities of an organization in general.

Some aspects of configuration management can be easily automated, especially those pertaining to revision and version control of documents and source code. A good discussion of configuration management tools in general, and those for personal computers specifically, can be found in [VAL87].

2.11 Software Project Management

Software projects, like any other project at Sandia, utilize valuable resources: time and people. And, like any other project, these resources need to be effectively managed in order to deliver a quality product on time. This section is concerned with project management, specifically the management of software development projects.

The key to any project's success is planning—the allocation and scheduling of time, people, and other necessary resources. The third volume of these Guidelines provides motivation for this planning, together with a recommended outline of a project management plan specifically for software projects. IEEE also provides an outline for use, *Standard for Software Project Management Plans* [IEE87a], as well as a companion guideline, *Guide for Software Project Management Plans* [IEE88a]. The project management plan should be used to document provisions to procure necessary manpower, establish dates to mark important milestones in the progress of the project, and clearly define project deliverables. Sandia-specific project guidelines for producing software for WR components can be found in *Process Guidelines for WR Software Development* [SCH88].

Just as important as allocation and scheduling of resources is the planning required for quality-related software activities. In the past, these activities were restricted to the testing phase, judging the software product much as one would inspect a widget rolling off of an assembly line. Current, recognized software engineering practices now incorporate quality activities into all development phases, building quality into the product rather than tacking it on at the end. So important is the planning for these activities that an entire volume of these Guidelines [SSGv1] is devoted to this issue.

In addition to the planning previously mentioned, the successful software project manager must have a grasp of the peculiar characteristics of software that differentiate its development from that of other products. For example, can one assess progress or productivity during a software development project? What are the pitfalls common to software projects, and how are they avoided? The answers to these and other questions lie outside the scope of this volume. Provided, however, are references to existing literature representing an enlightened view of software project management.

In the spirit of teaching others the lessons learned from one's own mistakes, Brooks [BRO75] published a much-cited book containing a number of short essays. These essays describe a variety of conceptual, as well as practical, problems encountered during one of the first large software development efforts. The advent of structured methodologies and other software engineering techniques introduces new issues which the software project manager must understand. Edward Yourdon, an acknowledged expert in the area of software development, has written a book that addresses these issues [YOU86]. Written in a clear, non-technical style, it is recommended reading for those managing software projects using structured techniques. Tom DeMarco, another recognized expert in the field, has authored a number of classics in the area of software development. One of DeMarco's books [DEM87] addresses projects and development activities of an intensely intellectual character. The subject matter is applicable to software development, as well as research and development activities that are common at Sandia.

3 Guidelines for the Selection of CASE Tools

As software development projects have become larger and more complex, it has become evident that methods of development that were adequate in the past are no longer sufficient. The size and complexity of many software projects now surpass the capabilities of a single person. Increased project size brings with it a disproportionate increase in software development costs and an accompanying need to moderate and control those costs. In addition, as software is used more frequently in safety and cost-critical applications, quality, reliability, and maintainability have become significant concerns. Over time, software developers have faced these concerns and gained more experience with large software development projects. One result is the evolution of systematic methods for planning and performing all facets of computer program development. These methods have been automated for speed and ease of use, and are implemented by *Computer-Aided Software Engineering (CASE)* tools.

3.1 Why Use CASE Tools?

The value of using software methodologies for development efforts has become evident. In general, these methodologies provide a well-defined way to develop software systems that are better designed and more reliable, maintainable, and economical. The benefits of using a methodology are derived from a “front-end” loading of the development process; more time and effort are expended during analysis and design than has been typical in the past. The increased emphasis on the early phases of software development requires a greater investment in the planning and design processes. The dividends include a reduction in the number of software defects and an earlier detection and correction of errors.

The automation of these techniques has made it possible to perform a number of iterations during the requirements and design phases, refining the analysis or design within a relatively short time. CASE tools save time and increase productivity. Without the use of CASE tools to perform this function, such a task would be performed manually or by hacking—tedious, time-consuming, error-prone approaches. Most CASE tools provide some capability to evaluate the internal consistency of analyses and designs produced. This capability eliminates a source of potentially serious defects in the final product. The use of CASE tools provides additional benefits in the areas of report generation and documentation. By using CASE tools during development, an automated means of demonstrating and reporting progress is available at any point during the project. Furthermore, software documentation is promptly available to support the finished product.

While most computerized software development tools have been designed to cover as many applications as practical, no one tool can be cited as the tool for all development efforts. The greatest advantages obviously result from choosing the tool best tailored to the type of software system being developed.

3.2 Considerations for the Selection of CASE Tools

In this section are posed sets of questions one might ask while investigating the appropriateness of a specific CASE tool. The list of questions included here is by no means complete. The intent of this section is to aid the potential CASE tool buyer or user when investigating a particular tool. It is hoped that these questions will serve to help the user ascertain if the tool in question fits project needs. Many of the issues raised by these questions are discussed in the next section.

Questions have been divided into six areas: capabilities, hardware requirements, licensing and copy protection, vendor support, graphics, and ease of use.

Capabilities

- What specific software development capabilities does this tool provide? What phases of the development life cycle does it support?
- Does the tool provide for any code generation? If so, what languages?
- Does the tool perform more than one function? If so, do the various functions communicate well? Is there adequate context checking across functions?
- Does the tool permit more than one user to work on a single project? If so, how is the project partitioned among users?
- If this is an integrated tool, is a data dictionary supported? Is direct access to the data dictionary possible? What kind of data consistency checks are done?
- Are configuration management and version control provided?
- Is it possible to import and export ASCII files? Does the tool support the creation and export of reusable libraries?
- Can macros be defined?
- Does a back-up capability exist? If so, is it adequate?
- Is this tool suitable for all sizes of problems? What is the largest problem it will adequately handle?

Hardware Requirements

- Does the tool run on a standard system for the machines supported? Can it easily be ported to another system of the same capability if necessary?
- How much memory does the tool require?

- Is special graphics hardware necessary?
- Is a mouse supported? Is it required?
- What printers or plotters are supported?
- Which monitors are supported? Is color an option?
- Are there any other special hardware requirements?

Licensing and Copy Protection

- What is the cost of this CASE tool?
- What method of copy protection is used, if any? Does it facilitate or hinder the backup of data?
- Does the license permit this CASE tool to reside on more than one machine? If so, does it employ a security-key system allowing only one machine to use it at a time?
- Does licensing restrict the tool from residing on a “cluster” of interconnected computers?
- Are site licenses available?
- Can Sandia’s standard license agreement be used? If not, are there any problems obtaining a license agreement?

Vendor Support

- Does the CASE tool vendor offer training in the use of the tool? What about tutorials or on-site classes?
- Is there a user hot-line? Does the vendor respond in a timely fashion?
- Is the hot-line contact well-informed and helpful?
- Can a maintenance agreement be obtained? Does the agreement include updates of the software?

Graphics

- If the CASE tool employs a graphic methodology, is the predefined set of graphic symbols complete?
- Can text (*e.g.*, labels, titles, IDs) be easily added? How easily can text be centered, changed, or moved?

- Does the order in which graphic images are created have significance? If so, can it be altered?
- How easily can the position of objects be changed? Does the same apply to labels, numbering, or interconnecting arcs?
- What context checking is performed as graphics are modified? Are inconsistencies brought to the user's attention, just disallowed, or not saved?
- If the tool employs hierarchical levels, how easily can one move between levels? Does a windowing system permit movement back to previous levels, or must they be redrawn? Is the time necessary to redraw the screen excessive?
- Can on-line help be accessed from within the graphics system?
- How much preliminary problem design must be done before one can use this tool?

Ease of Use

- *Is formal training necessary to use the tool effectively?*
- Does the accompanying documentation provide adequate support? Is it comprehensive, well-organized, indexed, and readable?
- Is the tool itself user-friendly? Does the tool employ menus, or is it command-driven?
- How easily can one move around within different sections of the tool? Can one exit easily?
- Is on-line help provided? Is the specificity of the help provided appropriate and useful?
- Does the tool use a consistent user interface across all functions? Is terminology meaningful and consistent?

3.3 What Type of Tools Are Needed?

The purpose of this section is to give those users unfamiliar with CASE tools some insight into the types of capabilities these tools possess.

CASE tools exist to cover most steps of the well-known waterfall life cycle model of software development. Some tools cover only one or two of these steps, while others integrate several individual tool functions into a single product.

3.3.1 Stand-alone Versus Integrated Tools

Stand-alone tools provide a single development or maintenance capability in the software life cycle. The better-known stand-alone tools cover requirements specification, user interface prototyping, configuration management, code analysis, and documentation. Stand-alone tools normally reside on a PC-level or VAX-level machine.

The term *integrated* refers to the combination of several individual tools into one system. The individual capabilities commonly found in integrated tools may include analysis and design functions to produce data dictionaries, data flow diagrams, data structure diagrams, entity relationship diagrams, structure charts, state transition diagrams, and control analysis matrices. These capabilities share an underlying data base which enables context checking to be performed across all development functions. For instance, redundant or conflicting data items and process definitions may be detected when the analyses and designs of a particular system are compared. Another common feature of integrated CASE tools is the availability of configuration management and version control covering all phases of development.

Currently, most integrated CASE tools are based upon the DeMarco and Yourdon approaches to structured analysis and structured design. One major variation in these tools depends on whether the tool includes provisions for the modelling and design of real-time systems. In some cases, real-time provisions are merely supplements to the system. In other cases, the tool is specifically designed as a real-time tool. Other variations in integrated tools include choices of style for graphic entities displayed, menu styles, colors, user interfaces, configuration management, and security. Most integrated tools allow for multi-user and multi-project use by providing a locking or password system for different portions of the project.

In general, integrated tools are more expensive than a corresponding aggregate of stand-alone tools. This added expense, and the additional functionality provided over a collection of stand-alone tools, can be traced to the cross-checking and data dictionary functions provided by the underlying data base. Some integrated tools require a workstation-level machine to provide the speed and power to adequately support the sophisticated graphics functions they utilize.

3.3.2 Specific Features of CASE Tools

Discussed below are features and enhancements that a user might look for in either stand-alone or integrated CASE tools.

3.3.2.1 Windowing

User interfaces that employ *windows* provide the user with an additional degree of power and flexibility. Windowing, the use of windows by a software tool, is essentially the

partitioning of the user's screen into different sections. Different parts of a program, or even different programs, can be displayed in each section. The size of these sections or windows can be altered so that some windows overlap others. Typical windowing operations allow windows to be hidden from view or shrunk into an icon or picture which can later be expanded. Window systems are usually controlled by the use of a mouse or other manual pointing device.

Windowing capabilities in CASE tools vary from simple systems to overlapping, user-configurable, multi-window systems. Some CASE tools simply provide several windows in a predefined, non-overlapping configuration. A more flexible windowing system allows the CASE tool user to display several levels or portions of the system into separate windows and view or modify them simultaneously. By altering the size of each window, the user can tailor the screen to display just the right amount of information in each window. If successive levels of a diagram hierarchy are maintained in separate, stacked windows, the user is spared the time required to save and then redraw the levels on the screen when moving between them.

Color may or may not be a feature of a particular tool. If color is available, its use can further enhance the windowing system by facilitating easy identification of stacked windows.

3.3.2.2 Menus

A significant part of the convenience and "user-friendliness" that a tool exhibits is attributable to its user interface. The manner in which the user is informed of and allowed to select options depends in large part on how those options are displayed and utilized. Menus provide a method for the user to choose from among the various options and to exercise those options without having to refer to documentation or memorize a list of options or commands.

Menu interfaces may or may not be available. Code analyzers, for example, typically require the user to enter strings of commands or instructions from the keyboard (*i.e.*, they are *command-line* driven). If available at all, a menu may be displayed only at the top or bottom of the screen. In this case, selections may be made using a mouse, by a keyboard-directed cursor, or by typing a number corresponding to the desired selection. In more sophisticated interfaces, a menu may "pull down" from the top of the screen or "pop up" on the screen. These types of menus are commonly associated with a mouse, allowing the user to "click on" the desired option.

Menus are often nested or organized in a tree-like hierarchy, with the first menu selection providing a second menu from which to select the desired function. The user may be able to move easily to desired levels in the menu hierarchy or may be forced to exit in reverse sequence through the hierarchy before making a new top-level selection.

3.3.2.3 Report Generation

The report generation capabilities of CASE tools vary widely. In some cases, the only output available may be a printable file predefined by the vendor in an ASCII format. On the other hand, more sophisticated, graphics-based systems may provide output formatted in one of the more popular printer definition languages, such as *PostScript*, or *pic*.

The most sophisticated integrated systems provide a query language to access an internal relational data base. Using the query language, report formats can be designed and printed according to the user's needs and required quality. However, the capabilities of these systems to produce reports are wide-ranging. Some systems are only capable of generating simple development verification and management reports; others, design documentation reports and user manuals. For projects required to adhere to specific DoD standards, there may exist options within the tool's framework to provide document audit functions. Interfaces to technical publishing systems or word processors may be provided.

3.3.2.4 Text Editing

Text editing capabilities should facilitate correcting rather than deleting and retyping. The ability to directly edit an ASCII file containing data, specification, or process definitions is an advantage. If label and process definition editing requires menu retrievals, the editing will normally take more time.

Interfaces to commonly used word processor packages may be available for editing long process specifications or data descriptions. A vendor-provided word processor may accompany an integrated system.

3.3.2.5 Graph Editing

Graph editing, a powerful and user-friendly capability, makes a significant difference in productivity. With a tool based on a graphic modelling or design technique, the user requires flexibility for correcting and expanding the graphic representation. Some tools require a number of menu requests to edit graphs. The more sophisticated tools use a mouse to move, insert, and delete objects and arcs in the graph. Using these tools, a developer can divide large graphs into several smaller linked graphs. Determining what capabilities exist for editing the tool-generated graph is important. Users should ascertain how changes made by them to previously generated graphics are handled by the error reporting mechanisms of the tool. A few of the CASE tools can generate data flow diagrams and/or structure charts from formatted functional specifications. When a tool has this capability, the effect that the changes to these graphical outputs have on the original formatted functional specification needs to be understood.

When a user modifies a graph, or moves from one graph to another, the computer must *redraw* the monitor screen. Redraw time becomes an important issue as the user continually updates, moves between graphs, and accesses information in other areas of the system. Windowing systems may alleviate this problem by allowing graphs to remain available, but hidden from view. Note the redraw time in any demonstration to determine whether it will be sufficiently fast on the user's system. Slow redraw rates are liable to become frustrating during development.

3.3.2.6 Data Dictionaries

Data dictionaries are basic to software engineering tasks and foundational to integrated CASE tools. Relatively inexpensive stand-alone products exist for developing data dictionaries to keep track of definitions of variables, data flows, and processes. The ability to reference variables back to processes and program modules is helpful during debugging and testing. The most advanced integrated tools use a data base management system (DBMS) together with a query language to support using the data dictionary. This data base can often be accessed separately from the rest of the tool. Report writing at any level of detail then becomes possible and easily implemented using query language macros. Investigate the specific DBMS used in the tool. While the user may be unfamiliar with the DBMS provided by the tool, it may be possible to transfer data to another readily available and familiar DBMS or spreadsheet.

3.3.2.7 Context and Level Checking

One of the main advantages of using integrated CASE tools is the coordination they provide between their different parts. These different parts may, for example, permit the creation and editing of data flow diagrams on one hand, and structure charts on the other. The specific editor for each part of the tool provides the means to create a hierarchy of levels for expanding processes, designs, or data structures into successively more detailed views.

Context checking, not only within the levels of one part of a tool but across different parts of a tool, is where integrated CASE tools show their greatest advantages. In advanced forms, this feature will check to confirm that data items and processes are defined, and that they are defined consistently within different parts of the tool. In a stand-alone data flow diagram tool, checking is performed across the expanded levels of the data flow diagram, data dictionary, and process specifications. An input to or an output from a process in any diagram must also appear in the lower levels used to further define that process. With an integrated tool, checking is expanded to include simultaneously all the different parts and levels of the tool (*i.e.*, all graphs and the underlying data base). For instance, the check should ensure that the data used in a data flow diagram is consistent with the definitions cited in the accompanying structure charts. Some tools require that data flow arcs be given unique

names, while others will allow the reuse of an arc name under certain circumstances. An example of this can occur when a data flow leaves a data store and is used as an input to different processes. Some tools allow data flow arcs to split or branch, using the same name. Others require that a separate name be used for the divided flow.

A variety of error and warning responses are provided to the user, depending on the specific tool being used. These responses range from disallowing the creation of a conflicting object, to printing a warning on the screen, or simply creating a new entry in the data base with none of the usual context connections cited. The degree of context checking could be selectable. Though different tools vary in their sophistication of checking provided, most tools will at least flag the creation of redundant processes and check that the input and output is consistent in all levels of a data flow diagram.

3.3.2.8 Code Generation

In some CASE tools, it is actually possible to generate source code based on functional and data definitions already created. Currently, the code generated is generally limited to data definitions and module declarations. The actual code produced may consist of a calling statement and the delimiters necessary to denote the end of the module, together with the declaration and typing of data being passed into the module. Little more is created than an empty shell that the user will later fill with the code necessary to perform the intended function. The code generated in this manner is sometimes called a *code frame*.

Ada, C, Pascal, and COBOL, are the most common languages for which code generation is provided. FORTRAN code generation will be available in the near future. Vendors are working toward the goal of being able to produce complete, finished code directly from the functional and data definitions contained in a data base developed by the user. The goal includes the expectation that structure charts and data flow diagrams will be produced along with the finished code.

3.3.2.9 Language Sensitive Editors

A Language Sensitive Editor (LSE) is a specialized text editor used to write computer code. Language sensitive editors provide the normal capabilities expected of an editor, but include selectable "templates" for specific programming languages. Other capabilities often found in LSEs provide diagnostic reviews of the code syntax, allow the user to define or modify template spacing and indentation, and provide on-line help as well. A few LSEs interface with common word processing packages. This facility permits text processing of module header paragraphs.

LSEs usually must be used with a particular version of compiler. When used with a compiler, compilation may be initiated while using the LSE to facilitate rapid error correction. A symbolic debugger may be provided, especially with LSEs on the larger systems (e.g., VAX). These editors usually contain multiple buffer capabilities and permit the use of split screens, but may not contain sophisticated windowing packages. Language sensitive editors are not usually included in integrated CASE tool packages.

3.3.2.10 Version Control

Version control is a function implemented by both integrated tools and stand-alone tools. Some degree of version control could also be provided by the operating system of the host computer or workstation. The version control features of CASE tools automate some configuration management functions, particularly those that are tedious and time-consuming.

Version control tools perform a number of functions related to configuration management. The most obvious function is the identification and maintenance of past software versions. Specialized stand-alone version control tools may also provide additional features. Features include methods to document the history of changes to different versions or to prevent unauthorized changes to particular versions.

The specific method used to store revisions may be of interest to a prospective user. Previous versions are stored and recreated in one of two ways. The simple way is to store entire versions, renaming each version with a unique name to identify the version. This, however, may consume large amounts of disk storage space for a large piece of software. The second technique overcomes this disadvantage by storing only one baselined version. Later versions are not stored in their entirety. Rather, only the changes to the baselined version are stored. The version control program then reconstructs the desired version by applying the stored changes to the baselined version when the user accesses the later version. While baselined versions have the advantage of conserving storage space, additional time lag must be expected when accessing previous versions.

3.3.2.11 Project Management

Project management tools are usually stand-alone tools used to support the scheduling and/or the budgeting of projects. These tools are not normally considered to be CASE tools since they aid in the management of all types of projects, not just software projects. Most of these programs will produce the graphics usually associated with project management activities such as Gantt charts, PERT charts, and resource plots. The more sophisticated of these tools permit the user to schedule resources for several projects at once.

The potential buyer of a project management tool should investigate a number of features. Among these are the maximum number of tasks allowed per project, the ease with which schedule deviations and holidays can be entered, the level of detail provided for resource assignments, error notification methods for resource or scheduling conflicts, and output capabilities. Special attention should be given to evaluating features that cannot be overridden. For example, a user may not want to use every form of chart notation provided by default, but may not be permitted to eliminate them. For some tools, the manner in which information is entered is cumbersome. Some project management tools provide an interface to directly enter data from a number of widely used spreadsheet programs. When selecting a project management tool, the user should first investigate his own needs carefully, and then compare the capabilities relative to the difficulty of using the package.

3.3.2.12 Reverse Engineering

Reverse engineering is the process of deriving design information from existing program source code. The ability to take any undocumented, poorly structured code and produce a structure chart or cross reference listing of its variables is invaluable for maintenance purposes.

The results typically produced by reverse engineering can include structure charts, data flow diagrams, variable listings and cross references, and module calls. While these results can be used to better understand and visualize old code, the capability to generate these results is still poorly facilitated by the tools which attempt it.

4 CASE Tools in Use at Sandia

This chapter contains reviews of CASE tools used at Sandia National Laboratories. The list of CASE tools described here is by no means complete; undoubtedly, new tools are constantly being acquired and reviews of tools already in use have not been supplied.

The intent of this chapter is twofold. The first is to show the different types of CASE tools being used at Sandia National Laboratories. The types of tools used in the Laboratory environment reflect the diversity of tools available in the commercial marketplace. The second is to provide a representative catalogue of these tools. It is hoped that disclosing the existence of these tools at Sandia will stimulate additional interest in CASE tools and encourage their use. Some of the tools listed in this chapter are available for use by other Sandians.

Before any CASE tool was listed in this chapter, it had to satisfy two prerequisites. First, each tool must currently be used in Sandia's software environment. Secondly, each tool must have an advocate or "sponsor"—someone willing to answer an occasional question about the tool. As this volume undergoes revision, new CASE tools will be added, and those no longer used deleted. In Appendix C is a form Sandians can use to describe newly acquired tools for inclusion in subsequent issues of this volume.

The listing of CASE tools described in this chapter is divided into six sections. Each section describes tools that are related by either their specific function or by their relationship to the software life cycle. These six sections describe tools used for user interface prototyping, the analysis, design, and/or implementation of software systems, software configuration management, static code analysis, reverse engineering, and project management. The sections appear in the order listed above. Within each section, the tools are ordered alphabetically according to the tool's name.

Information regarding each tool is given in a uniform format that provides a brief description of the tool's function, the environment in which it runs, and the name and address of the tool's vendor. In addition, comments provided by the tool contributors have been summarized and categorized into *Pro* and *Con* sections.

4.1 Tools for User Interface Prototyping

The CASE tools described in this section are used for user interface prototyping. The interested reader should refer to the overview of user interface prototyping contained in Chapter 2 of this volume.

Tool Name: *Dan Bricklin's DEMO, Version 1A*

Tool Type: *User Interface Prototyping Tool*

Description:

- *DEMO is a developer's user interface prototyping tool*
- *Uses command windows, prompts, menus/messages, and function keys to create and view a series of views or slides on the screen*
- *Slides can be "run" by DEMO under automatic control, simulating a program*
- *Prototypes developed accept user input from the keyboard to make selections (e.g., menu selections)*

Environment:

- *Runs on an IBM PC/XT/AT*
- *Supports use of CGA and EGA graphics*

Vendor Information:

*Software Garden, Inc. (617) 332-2240
P.O. Box 373
Newton Highlands, MA 02161*

Approximate Cost: \$75

User Comments, Pro:

Comes with a tutorial and tutorial diskette. Product well documented. Can create up to 50 copies of demonstration programs that can be distributed freely without the main DEMO program.

User Comments, Con:

Somewhat confusing to use when first learning the program. Data entered on one screen cannot be carried over to others (i.e., program does not provide way to access field entries from other screens).

Additional Information: *John Franklin, Organization 7254*

Tool Name: *MIRAGE, Version 1.54*

Tool Type: *User Interface Prototyping Tool*

Description:

- *MIRAGE is a rapid interface prototyping tool that allows designers to interactively mock-up human-computer interfaces*
- *Provides a hierarchical menu-driven system that provides interface development independent of any programming knowledge*
- *Allows developers to rapidly build displays and establish paths between displays*
- *Provides immediate simulation of the prototyped user interface system*

Environment:

- *Requires IBM PC/XT/AT or compatible*
- *Minimum of 256K RAM, optimal performance using a RAM disk*
- *Supports monochrome and/or color monitors*
- *Requires either a touch screen or mouse for designing a simulation (neither is required to view a simulation)*
- *Supports keyboard devices*

Vendor Information:

Developed by Jim McDonald of New Mexico State University under contract to Sandia. Distributed free of charge within Sandia environment.

User Comments, Pro:

Extensive user interface development can be done without any programming knowledge. Minimum system familiarization time required (1.5-2 hours). Provides a means of integrating human factors and other support functions in a timely and cost effective manner. Menu-driven system makes design and simulation easy.

User Comments, Con:

The prototype cannot be used in the system directly at this time.

Additional Information: *Doug Mangum, Organization 2312*
Betty Chao, Organization 2315

4.2 Tools for Analysis, Design, and Implementation

The CASE tools described in this section are used for the analysis, design, and implementation of software systems. It would be difficult, if not impossible, to separate the descriptions of tools in this section into three separate sections representing each function. Sometimes two, if not all three, of these functions have been integrated into a single tool.

Most of the CASE tools described in this section perform structured analysis and design using the DeMarco/Yourdon methodologies discussed in Chapter 2. There exist, however, those tools that provide additional capabilities in the form of information modelling, real-time system analysis, and software implementation, either in a stand-alone manner, or integrated together with other functions.

Tool Name: *Ada Design Language (ADADL)*

Tool Type: *Ada Design and Reverse Engineering Tool*

Description:

- *Receives Ada code as input, generates both preliminary and detailed design documentation as output*
- *Output of this tool can be used as input to AGT program (see description of AGT in Section 4.5)*
- *Output documentation includes cross-reference reports, a data dictionary, and design hierarchy trees*
- *Supports object-oriented design*
- *Suitable for all sizes of projects*

Environment:

- *Unix-based systems*

Vendor Information:

*Software Systems Design (714) 625-6147
3627 Padua Avenue
Claremont, CA 91711*

Approximate Cost: \$6000 – \$13,000 per CPU

Site licenses available at nine times the single CPU cost

User Comments, Pro:

Vendor is responsive to our needs and suggestions.

User Comments, Con:

Not enough control over "pretty printing". Does not generate Postscript output.

Additional Information: *Amelia Maxted, Organization 9224*

Tool Name: *AIDA, Version 1.0*

Tool Type: *Implementation Tool for Real-Time Systems*

Description:

- *AIDA is a package of tools to facilitate creation of real-time, multi-tasking applications in the "C" language*
- *Provides multi-process control, error trapping, interprocess communications, text screen design (including menus), and testing tools to build and debug an application very quickly*
- *Allows macro definition and library creation by user*

Environment:

- *Runs on IBM PCs and compatibles using CGA or EGA graphics*
- *IBM ARTIC coprocessor card for data interface is supported*
- *Can be ported to any machine that has a full implementation of the "C" language*

Vendor Information:

Sandia-developed tool

No cost to Sandia users

Training provided by Organization 5268

User Comments, Pro:

Considered user-friendly to those knowledgeable about software engineering.

User Comments, Con:

Training is advisable. Only limited documentation is available at present.

Additional Information: *Rodema Moseley, Organization 5268*

Tool Name: *Analyst/Designer Toolkit*

Tool Type: *Structured Analysis and Design Tool*

Description:

- *Allows user to create and edit data flow diagrams, state transition diagrams, entity-relationship diagrams, structure charts, and presentation diagrams*
- *Has an integrated data dictionary that supports all of the above applications*
- *Provides a user-friendly, mouse driven environment*
- *Large variety of hard-copy data dictionary reports can be generated*
- *Allows library creation and macro definition by user*
- *Permits import and export of ASCII files*

Environment:

- *IBM PCs and compatibles*
- *Requires 640K RAM and 10 Mbytes of disk space*
- *Supports Epson FX series printers, HP plotters, IBM Proprinter, and HP Laserjet (via conversion program)*
- *Supports Monochrome, CGA, and EGA monitors*

Vendor Information:

*John Roche
Yourdon, Inc. (415) 871-2800
1501 Broadway
New York, NY 10036*

Approximate Cost: \$2000

User Comments, Pro:

Quite powerful for the price. There is no limitation to the number of drawings (charts) in a given design.

User Comments, Con:

Printing diagrams using the HP Laserjet is very awkward.

Additional Information: *Ed Nuckolls, Organization 2311*

Tool Name: *CASE Analyst/RT*

Tool Type: *Real-Time Structured Analysis Tool*

Description:

- *A graphics-oriented structured analysis tool for generating, editing, and plotting data flow and control flow diagrams*
- *Data dictionary can be automatically generated or updated from the data/control flow diagrams, and edited using any VMS text editor*
- *Skeletal process specification can also be generated*
- *Table Editor provided to described state event matrices and other decision tables*
- *Supports either Ward/Mellor or Hatley/Pirbhai real-time methodology (desired method is selected by user at run-time)*
- *Provides context checking in data flows between parent and child diagrams, and among the diagrams, dictionary, and specifications*

Environment:

- *Runs in VAX/VMS environment*
- *Supports Tektronix graphics terminals*
- *Supports Epson, QMS Laser, LN03+, and LaserWriter (Postscript) printers*
- *Available at Sandia on SAV35*

Vendor Information:

*Mentor Graphics Corporation (800) 547-4303
8500 S.W. Creekside Place
Beaverton, OR 97005*

Cost of software license depends on host computer and ranges from \$11,500 to \$74,000

User Comments, Pro:

The most useful features of this package are the data flow design tools, the evaluation reports, and the data dictionary generated and updated automatically from the data flow diagrams. Mentor has made use of existing VAX/VMS facilities where possible, eliminating a major effort to learn new text editors and file management systems.

References: *[WAR85], [HAT87]*

Additional Information: *Michael Sharp, Organization 5145*

Tool Name: *IDMS/Architect, Release 1.0*

Tool Type: *Structured Analysis Tool*

Description:

- *Used to produce data flow diagrams, entity-relationship diagrams, Bachman diagrams, and others*
- *Integrated data dictionary*
- *Aids data modelling by assisting in data normalization*
- *Provides mechanism for design verification*

Environment:

- *IBM PCs and compatibles*
- *Requires 640K RAM, 12 Mbytes of disk space*
- *Compatible with most dot matrix printers; laser printers requires an additional utility*

Vendor Information:

*Ken Pulvino (505) 889-0042
Cullinet
6121 Indian School Road, NE
Albuquerque, NM 87110*

Estimated Cost: \$8000

Maintenance contract available

User Comments, Pro:

On-line help available. Good graphics capability. Integrated data dictionary provides a centralized repository for design definition, thus allowing comprehensive reports and elimination of redundant keying of information. Product is useful even if IDMS data base is not the data storage mechanism.

User Comments, Con:

Adding customized icons to graphics is difficult. Runs slow on a PC-XT. Hard to coordinate changes among many developers.

Additional Information: *Ken Osburn, Organization 2821*

Tool Name: *PC-IAST, Version 2.10D*

Tool Type: *Information Modelling/Analysis Tool*

Description:

- *Analysis tool using the NIAM information modelling technique*
- *Takes data relationships as input, forms tables for data base in 5th normal form as output*
- *Can output SQL necessary to produce tables*
- *Assists library creation and macro definition by user, capable of importing/exporting ASCII files*
- *Creates English language description of relationships from relationship diagrams*

Environment:

- *IBM PC or compatibles*
- *Requires a minimum of 256K RAM and hard disk for secondary storage*

Vendor Information:

*Andy Rutan (505) 262-5030
Control Data Corp.
300 San Mateo Blvd N.E.
Albuquerque, NM 87108*

Estimated cost: \$7000

User Comments, Pro:

PC-IAST will come up with recommended tables in 5th normal form, a good starting point for creating data bases. Program is user-friendly.

User Comments, Con:

Must be familiar with NIAM modelling methodology to use. Documentation is not sufficient to learn methodology, only to use the tool.

References:

Nijssen, G.M., "A Framework for Advanced Mass Storage Applications", Proceedings IFIP Medinfo 1980, Tokyo, 1980.

Additional Information: *Norm Stevens, Organization 2825
Al Beradino, Organization 2812
John Sharp, Organization 2825*

Tool Name: *Prokit*Analyst, Version 2.0*

Tool Type: *Structured Analysis and Design Tool*

Description:

- *PC-based design tool for software system designers and developers*
- *Assists in developing data flow diagrams, while automatically supporting data dictionary documentation*
- *Provides completeness checks of data entities (flows, stores, processes, elements, structures, etc.)*
- *Multi-user security provided*
- *Suitable for different sizes of problems*

Environment:

- *IBM PC/XT/AT and compatibles*
- *Copy protected*
- *Requires 640K RAM, 10MB disk space, DOS 3.x, and Math Coprocessor*

Vendor Information:

*McDonnell-Douglas Information Systems Group (800) 325-1087
Mail Stop L864-280-2
P.O. Box 516
St. Louis, MO 63166*

*Site licenses and maintenance contracts available
Approximate Cost: \$2500*

User Comments, Pro:

Easy to use for someone familiar with software engineering methods. Provides very complete system documentation and analysis information with minimal effort. Has a moderate "learning curve". The package is consistent with the MIS (management information systems) being taught in the universities.

User Comments, Con:

Current version does not work easily with Prokit data generated on other PCs. It also is a memory hog and requires a very powerful PC to generate the required graphics quickly.

Additional Information: *Robert Banwart, Organization 131
Donna Campbell, Organization 2625
Ramona Gauna, Organization 151*

Tool Name: *SA Tools*

Tool Type: *Structured Analysis Tool*

Description:

- *Enables creation, evaluation, and modification of structured analysis documents*
- *Includes a graphics editor, text editor, an evaluation program, a fix program, and a display program*
- *Creates data flow diagrams, data dictionaries, and mini-specifications using the DeMarco methodology*
- *Suitable for all sizes of projects*

Environment:

- *Runs on VAX under VMS 4.x and UNIX*
- *Mouse is supported but not required*

Vendor Information:

*Mentor Graphics Corporation (800) 547-4303
8500 S.W. Creekside Place
Beaverton, OR 97005*

User Comments, Pro:

Good tutorial is provided in the User Manual. The tool requires minimum familiarization time. A user can become reasonably proficient in 2-3 hours of use.

User Comments, Con:

The Graphics Editor is not as versatile as those in some of the competitive software because it has a small vocabulary.

Additional Information: *Amelia Maxted, Organization 9224
Suzanne Rountree, Organization 2812
Randall W. Simons, Organization 9224*

Tool Name: *SDM/Structured*

Tool Type: *Forms-driven, paper-based systems development methodology*

Description:

- *SDM is a series of manuals that detail the phases of the systems development life cycle (is not a computer-based tool)*
- *Provides the user with project and phase estimating worksheets and guidelines, phase-by-phase life cycle tutorials, and comprehensive task-level assistance*
- *Can be used from the project feasibility stage through the maintenance phase*
- *Implements requirements specifications, structured analysis and design, testing, and maintenance*

Vendor Information:

AGS Management Systems

Approximate Cost: \$40,000

Site license and maintenance contract available

User Comments, Pro:

The product has good tutorial and documentation features. SDM is currently being used as the 2620 software development methodology.

User Comments, Con:

SDM is somewhat overwhelming in sheer size of the manuals. Very form, report, and paper-oriented.

Additional Information: *J.R. Schofield, Organization 2624*

Tool Name: *Software Through Pictures*

Tool Type: *Integrated Structured Analysis and Design Tool*

Description:

- *An integrated environment for the early stages of software development*
- *Includes a set graphical editors and supporting programs that aid in the analysis of requirements, software design, and prototyping*
- *Code generation of data definitions in C, Pascal, and Ada*
- *Direct access to underlying data base is provided*
- *Open system architecture that allows modification and extension of the system*
- *Provides multi-user security, allows macro creation and library creation by user*

Environment:

- *VAX, Sun, and Apollo workstations*

Vendor Information:

*Interactive Development Environments, Inc. (714) 851- 0511
150 Fourth Street, Suite 210
San Fransisco, CA 94103*

Approximate Cost: \$20,000

User Comments, Pro:

Extremely user-friendly and consistent across all tools. Very powerful and flexible.

User Comments, Con:

Some difficulty installing the newly released VMS version due to bugs and omissions in installation instructions.

References:

Forman, Betty, "Designing Software With Pictures", Digital Review, July 11 1988, pp. 39-42.

Additional Information: *Margaret Olson, Organization 2814
Rodema Moseley, Organization 5268*

Tool Name: STATEMATE

Tool Type: Real-time Analysis and Design Tool

Description:

- *Allows data base access by all project members with normal input through graphic and forms editors*
- *Uses hierarchical methodology suitable for very large problems*
- *Consists of three distinct elements: Analyzer, Documentor, and Kernel*
- *The Kernel*
 - *Provides three graphical editors for defining and modelling real-time software and/or hardware systems:*
 - *State Charts Editor, used to define and model the dynamic, behaviorial view*
 - *Activity Charts Editor, used to define and model the functional view with functional flow*
 - *Module Chart Editor, use to define and model the structural view of the system organization*
 - *The editors are integrated into a relational data base manager which maintains the life cycle dictionary and provides syntax checking and basic consistency checking of the relational data base*
 - *Can generate prototype Ada code with the optional Code Generator*
- *The Analyzer*
 - *Provides a powerful analysis language for interactive or batch mode simulation and analysis of STATEMATE specified systems*
 - *Incorporates a simulator which allows the user to examine the dynamic response of the system being developed to external stimuli*
 - *Includes a dynamic testing tool that examines the data base of the system being developed for consistency, completeness, and reachability*
- *The Documentor*
 - *A configurable system for automatically generating the documentation of a system specification*
 - *Templates for MIL-STD-2167A documentation are provided*

Environment:

- *Runs on VAX and Sun workstations with 6 Mbytes RAM (10 Mbytes preferred)*
- *Requires hi-resolution graphics monitor*
- *Requires installation of license key*
- *Analyzer and Documentor elements cannot be run without the Kernel*
- *The Documentor supports: Printers - DEC LA100, LA150, LA50;
Epson FX80, FX100
Plotters - Hewlett Packard*

Vendor Information:

*Joel Brill (714) 843-9416
i-Logix, Inc.
22 Third Avenue
Burlington, MA 01803*

*Approximate cost per stand-alone workstation: Kernel - \$10,000
Analyzer - \$25,000
Documentor - \$10,000*

User Comments, Pro:

Good support, applications engineer assigned to each installation. On-site training included with software. Complex system, but documentation is thorough. Some of the methodology is unique to STATEMATE.

Kernel:

The basic consistency checking is actually quite extensive. Editors are fairly easy to use and powerful. The graphics are of good quality.

Analyzer:

Provides the capability for checking system specifications that few other design automation tools can provide. In interactive mode, the simulation tool can use the graphics editor to display the results of the simulated actions.

User Comments, Con:

The system is large, using considerable disk and processor resources. The modelling concepts used in this tool require considerable time and effort to learn. No on-line help is provided. The software is relatively expensive and not well-suited to specifying data processing systems. Documents produced require editing for some purposes. The simulation and dynamic test languages require significant time and effort to learn.

References:

Harel, D., et al., "On the Formal Semantics of Statecharts," Proceedings 2nd Symposium on Logic in Computer Science, 1987.

Hughes, David, "Company Uses Embedded System Specification Tools on LaviWork," Aviation Week and Space Technology, February 15, 1988.

Schindler, Max, "Real-time Software-design Tool Combines Three Different Views," Electronics Design, July 23, 1987.

Additional Information: *Norm Kolb, Organization 2336
Steven Richards, Organization 2336*

Tool Name: *Teamwork*

Tool Type: *Structured Analysis and Design Tool*

Description:

- *A tool for developing structured analysis and design*
- *Allows development of entity-relationship diagrams, data flow diagrams*
- *Uses common data dictionary to support above methods*
- *Supports real-time systems analysis*
- *Provides multi-user access to data base*
- *Allows export of files to Interleaf and other graphics/text editors*
- *Suitable for all sizes of problems*

Environment:

- *Runs on HP9000, PS/2, RT/PC, VAX, Sun, and Apollo workstations*
- *Requires 6 Mbytes RAM, 16 Mbytes disk space with data base*

Vendor Information:

*Cadre Technologies, Inc. (401) 351-5950
222 Richmond St.
Providence, Rhode Island 02903*

Approximate Cost: \$9500 per CPU, \$45,000 and up for network and cluster licenses

User Comments, Pro:

Supports Shlaer-Mellor Object-Oriented Analysis methodology. Cadre is working to enhance product to further support this methodology. The Buhr/Ada tool is available.

User Comments, Con:

Somewhat slow due to data base accesses. Have used better picture drawing tool—awkward to make all lines straight and keep component sizes consistent.

Additional Information: *Jeffrey Kern, Organization 9221*

Tool Name: *Teamwork/PCSA, Version 3.00*

Tool Type: *Structured Analysis - Data Flow Diagramming Tool*

Description:

- *Interactive PC-based Structured Analysis tool using DeMarco methodology*
- *Can create data flow diagrams, data dictionary, and process specifications*
- *Checks consistency and balance of data flow diagrams*
- *Capable of producing hardcopy set of documents*
- *Can import/export ASCII files*

Environment:

- *Runs on IBM PC and compatibles, requires at least 512K of memory, a mouse, and graphics adapter*
- *Supports Apple Laser, Epson FX, and HP Laserjet, and Postscript printers*
- *Not copy-protected, but requires hardware protection block to run software*

Vendor Information:

*Cadre Technologies, Inc. (401) 351-5950
222 Richmond St.
Providence, Rhode Island 02903*

Cost: \$995 each, \$525 for 25 copies or more

User Comments, Pro:

Easy to use, with pop-up menus at appropriate locations. Hot-line available for questions, judged to be useful. Files created with PCSA can be uploaded to more powerful workstation-based Cadre Teamwork.

User Comments, Con:

No on-line help (although most commands are self-explanatory).

References: *[DEM79]*

Additional Information: *David Harris, Organization 5173*

4.3 Tools for Software Configuration Management

Tools listed in this section provide a way to implement some of the requirements for a software configuration management scheme. The interested reader should refer to the overview of software configuration management contained in Chapter 2 of this volume.

In particular, configuration management tools can be used to archive previous versions of software and its documentation, coordinate or prevent changes by different developers, and provide a version history and record of changes.

Tool Name: *DEC/CMS (Code Management System)*

Tool Type: *Configuration Management Tool*

Description:

- *Stores ASCII source files in a library*
- *Maintains a history of all user access to the files*
- *Produces a new version of a file when the file is changed*
- *Retains all versions of a source file and permits later access to the requested version*
- *Can be used with DEC/MMS to provide configuration control*

Environment:

- *Runs in VAX/VMS environment*
- *Available at Sandia on SAV02, the distribution point node*

Vendor Information:

*Bill Bartko (505) 761-2730
Digital Equipment Corporation
5600 Jefferson, NE
Albuquerque, NM*

Right-to-copy license required, price depends upon the host computer

User Comments, Pro:

CMS is user-friendly. Easy to learn to use CMS from the documentation. On-line help is available. Very good for writing and maintaining code. Any version of a file is accessible. Use of groups and classes allows organization of CMS files for easy use. Good change control is provided even with many users working with the same files.

User Comments, Con:

Only ASCII source files can be placed in a CMS library. At compile time, files included in other source files must normally be fetched from the library.

References: *VAX DEC/CMS Reference Manual, User's Manual*

Additional Information: *Sylvia Jean-Louis, Organization 2634
Cheryl Haaker, Organization 2634*

Tool Name: *DEC/MMS (Module Management System)*

Tool Type: *Configuration Management Tool*

Description:

- *A software management tool that automates the building of software systems*
- *Executable software modules are constructed from components stored in a common source library*
- *Requires a description file and the logical dependencies to define the software module*
- *Can be used with DEC/CMS to provide configuration control*

Environment:

- *Must be run on VAX/VMS Version 3.4 or later*
- *Available at Sandia on SAV02, the distribution point node*

Vendor Information:

*Bill Bartko (505) 761-2730
Digital Equipment Corporation
5600 Jefferson, NE
Albuquerque, NM*

Right-to-copy license required, price depends upon the host computer

User Comments, Pro:

MMS is user-friendly with on-line help and good documentation. MMS is powerful. Once the MMS description file is correctly set up and all dependencies shown, the processing steps, compiling, linking, etc., will be done automatically. The description file and dependencies are easily changed. A great advance compared to doing things by hand. MMS allows users to build/rebuild their systems or programs with one command line once the MMS description file and dependencies are set up.

User Comments, Con:

Some kind of introduction to MMS would be useful. Using MMS can be very confusing. The MMS description file must be set up with care so that all required dependencies are shown. When dependencies change, the MMS description file must also reflect the changes.

References: *VAX DEC/MMS Quick Reference Guide, User's Guide*

Additional Information: *Sylvia Jean-Louis, Organization 2634
Cheryl Haaker, Organization 2634*

Tool Name: *Librarian, Version 3.7*

Tool Type: *Configuration Management Tool*

Description:

- *Is a Source Code Manager and Control tool*
- *Also a control language, load library, and tracking and reporting tool*
- *Provides multi-user security*
- *Allows library creation by users*

Environment:

- *IBM MVS environment*

Vendor Information:

*Computer Associates International, Inc. (703) 821-1700
8300 Greensboro Drive, Suite 700
McLean, VI 22102*

Site licenses and maintenance contracts available.

Approximate Cost: \$50,000+

User hot-line available

User Comments, Pro:

Improved user interface over previous versions. Data compression routines save up to 70% of required storage space. Easy to use, on-line help available.

User Comments, Con:

Limited applicability unless establishing an IBM MVS production environment. This company may be bought by another company soon.

Additional Information: *J.R. Schofield Jr., Organization 2624*

Tool Name: *Polytron Version Control System (PVCS)*

Tool Type: *Configuration Management/Version Control*

Description:

- *Configuration management tool for PC-based systems*
- *Suitable for projects of all sizes*
- *Used to track and maintain different versions of software*

Environment:

- *Runs on IBM-PCs and compatibles*
- *Supports a large variety of monitors and printers*
- *VAX and network versions also available*

Vendor Information:

*Polytron Corporation (503) 645-1150
1700 N.W. 167th Place
Beaverton, OR 97006*

Approximate Cost: \$395

User Comments, Pro:

A big help in preventing the use of old source code, and for two individuals working on code at the same time.

References: *[VAL87]*

Additional Information: *Larry Desonier, Organization 5246*

Tool Name: *Source Tools, Version 1.2A*

Tool Type: *Configuration Management Tool*

Description:

- *A file management system that stores files, records revisions, and retrieves specific versions on demand*
- *Builds large systems from smaller components and performs file comparison*
- *Coordinates changes made by several users as they develop modules, then recreates programs affected by those changes*
- *Creates audit trails by user ID, comments, time and date stamps*

Environment:

- *Supported on IBM-PCs, VAX, PDP-11*

Vendor Information:

*Oregon Software, Inc. (800) 874-8501
6915 SW Macadam Ave.
Portland, OR 97219*

Approximate Cost: \$7500

Additional Information: *Bruce Malm, Organization 5173*

4.4 Tools for Static Code Analysis

The CASE tools listed in this section have been classified as *static analysis* tools. Such tools are used to analyze source code for various properties, such as complexity or correctness of syntax and module interfaces. These tools are typically used during the implementation and maintenance phases of software development and use.

The reader who is interested in software complexity should refer to the overview of software metrics contained in Chapter 2 of this volume.

Tool Name: *ANALYZ, Version 1.10*

Tool Type: *FORTRAN Static Analysis Tool*

Description:

- *Provides static analysis of FORTRAN programs to detect inefficient and erroneous coding*
- *Lists inconsistent data types in formal parameters, variables defined but not used, variables used but not defined, differences in COMMON blocks, etc.*
- *Written for VAX FORTRAN, but can be (and is) used to check FORTRAN written for other compilers*
- *Suitable for all-sized problems*

Environment:

- *Runs under VAX/VMS*

Vendor Information:

*Computer Sciences Corp. (505) 242-3131
Suite 200
2100 Air Park Road S.E.
Albuquerque, NM 87106*

Available free to U.S. Government agencies, already at Sandia

User Comments, Pro:

Easy to use. Checks for errors not often detected by FORTRAN compilers.

User Comments, Con:

Cannot list errors without a complete FORTRAN source listing (scheduled to be optional in a future release). Requires significant disk space for working and output files.

Additional Information: *Arnold Elsbernd, Organization 2614*

Tool Name: *FORTRAN-LINT, Version 2.54*

Tool Type: *FORTRAN Static Analysis Tool*

Description:

- *Static analysis of FORTRAN programs across modules to detect inefficient and erroneous coding*
- *Written for VAX FORTRAN, but can be used to check FORTRAN code for other computers*
- *Detects inconsistent data types in formal parameters, variables used but not defined, variables defined but not used, differences in COMMON blocks, etc.*
- *Suitable for all sizes of programs*

Environment:

- *Runs under a VAX/VMS environment*

Vendor Information:

*Information Processing Techniques Corp. (415) 494-7500
1096 East Meadow Circle
Palo Alto, CA 94303*

*Approximate Cost: \$4000 per CPU (license required)
Sandia has site license for 15 VAXs*

Maintenance contract maintained by Division 2614

User Comments, Pro:

Easy to use, no special training required. Checks for errors often not found by the FORTRAN compiler.

User Comments, Con:

Requires significant disk space for working files and output files. Program output not adequate for producing hardcopy documentation.

Additional Information: *Arnold Elsbernd, Organization 2614*

Tool Name: *JCLCHECK, Version 5.2*

Tool Type: *Job Control Language Checker*

Description:

- *Reduces run-time errors by checking the correctness of job control language (JCL) statements prior to execution*
- *Verifies the existence of required data set files*
- *Provides error detection, data set blocking efficiency, standards enforcement, and report generation*
- *Reports can be generated for jobstreams, merged procedures and JCL, data set and program cross-referencing*
- *An additional feature, SYSCHECK, validates every job submitted for execution*
- *Many features can be invoked from within ISPF editor*

Environment:

- *Any IBM 43xx MVS computer*

Vendor Information:

*Computer Associates International, Inc. (703) 821-1700
8300 Greensboro Drive, Suite 700
McLean, VA 22102*

Approximate Cost: \$25,000

Site license and maintenance contract already at Sandia

User Comments, Pro:

Very impressive to a novice. Useful for experienced JCL programmers and batch production environments. Especially useful for staff who are casual or non-expert level users.

References: *Installation Manual; User's Guide*

Additional Information: *J.R. Schofield, Organization 2624*

Tool Name: *MAT (Maintainability Analysis Tool), Version 12.01*

Tool Type: *FORTRAN Static Analysis Tool*

Description:

- *Static analysis of FORTRAN programs across modules to detect inefficient and erroneous coding*
- *Written for use on VAX computers, but can be used to check FORTRAN code for other computers*
- *Detects inconsistent data types in formal parameters, variables used but not defined, differences in COMMON blocks, etc. Messages can be switched off if desired*
- *Suitable for all sizes of programs*

Environment:

- *Runs under a VAX/VMS environment*
- *Requires 400 blocks of disk space*

Vendor Information:

*Gerald M. Burns
Science Applications International Corporation (703) 979- 5910
1213 Jefferson Davis Highway, Suite 1500
Arlington, VA 22202*

Approximate cost: None, Sandia has site license for all VAXs

Central site maintenance contract

User Comments, Pro:

Easy to use, no special training required. Checks for errors often not found by the FORTRAN compiler. Help available from Arnold Elsbernd.

User Comments, Con:

Program output not adequate for producing hardcopy documentation. No on-line help available.

Additional Information: *Arnold Elsbernd, Organization 2614*

Tool Name: *S__PAT/CAM*

Tool Type: *Complexity Analyzer*

Description:

- *Analyzes code for logic function, syntax errors, decision nodes, branches, interrupts, and module complexity based on McCabe's cyclomatic complexity metric*
- *Identifies critical test paths for a module and graphs the module's logic*
- *Prints and graphs all linearly independent test paths of the module*

Environment:

- *IBM PC/XT/AT or compatible*
- *DOS 3.x*
- *Version site licensed to Sandia currently works with Hewlett-Packard BASIC code*

Vendor Information:

*McCabe and Associates (301) 596-3080
Twin Knolls Professional Park
5501 Twin Knolls Road
Suite 111
Columbia, MD 21045*

User Comments, Pro:

Easy to use. CAM is useful for determining test paths to exercise during the test phase of development. The product provides a complexity metric, so the user can judge when maintainability improvements have occurred.

User Comments, Con:

Current Sandia version was contracted only to work with code written in Hewlett-Packard RMB BASIC, Version 3.0.

References: *[MCC76]*

Additional Information: *Darl Patrick, Organization 2854
Gene Bowling, Organization 7252*

4.5 Tools for Reverse Engineering

Reverse engineering tools are typically used during the maintenance phase of software use. In general, reverse engineering tools take as input source code listings and generate information relating to the structural design of the code. Such information may describe calling hierarchies of the modules within a program, where program variables are used, and even graphic information such as structure charts.

Tool Name: *Ada Graphical Tool (AGT)*

Tool Type: *Reverse Engineering Tool for Ada Code*

Description:

- *A prototype tool developed at Sandia to produce graphic illustrations of code design*
- *Generates WITH-HIERARCHY and DECLARATION diagrams from code (see reference)*
- *Uses the output from ADADL program (see preceding description in this section) to produce structure charts, WITH-HIERARCHY charts, and DECLARATION diagrams*
- *Produces graphic output in GKS meta-file format, which may be graphically edited by using GRED (see description which follows in this section)*

Environment:

- *Runs on Unix-based systems*
- *Supports Postscript and Tektronix 401x and 411x compatible printers*

Vendor Information:

Sandia-developed tool

User Comments, Pro:

Reverse engineering enforces agreement between code and graphics, aiding both design and maintenance

User Comments, Con:

This is a prototype software tool, requiring both UNIX and ADADL. AGT is an unsupported, "home-grown" product. If you need something fixed, you may have to fix it yourself.

References:

Maxted, Amelia and John C. Rowe, "An Ada Graphical Tool to Support Software Development," Ada Letters, Using Ada, ACM SIGAda International Conference, Dec. 8-11, 1987.

Additional Information: *Amelia Maxted, Organization 9224*

Tool Name: *GRaphical EDitor (GRED)*

Tool Type: *Graphics Tool for Use with AGT*

Description:

- *A graphics editor that can be used to generate and edit any GKS meta-file*

Environment:

- *Requires a Unix and C environment*
- *Supports Postscript and 4014 compatible printers*

Vendor Information:

Sandia-developed tool

User Comments, Pro:

GRED supports picture editing with a rich set of commands. Source code for GRED is available, making it possible to modify it yourself.

User Comments, Con:

GRED is an unsupported, "home-grown" product. If you need something fixed, you may have to fix it yourself.

References:

Ground System Development Manual, Division 9224, GRMN(3), INTERPGMT(1), GRED(1).

Additional Information: *Randall W. Simons, Organization 9224*

Tool Name: *Tree Diagrammer*

Tool Type: *Reverse Engineering Tool*

Description:

- *Reads input source code listing and produces an "organization chart" showing program structure*
- *Show hierarchy of calls to functions, procedures, and subroutines*
- *Indicates recursive calls*
- *Works with C, BASIC, Pascal, dBASE, FORTRAN, and Modula-2*

Environment:

- *IBM PCs and compatibles*
- *Requires minimum of 256K RAM*

Vendor Information:

*Aldebaran Laboratories (415) 934-4395
3339 Vincent Rd.
Pleasant Hill, CA 94523*

Approximate Cost: \$77

User Comments, Pro:

A powerful tool in analysis of code structure and also for debugging. Useful for documentation purposes.

Additional Information: *Larry Desonier, Organization 5246*

4.6 Tools for Project Management

The tools cited in this section are not CASE tools per se, but are useful for the management of software development projects. These tools implement techniques for project scheduling and the allocation of resources. An overview of software project management is included in Chapter 2 of this volume.

Tool Name: *Harvard Total Program Manager II, Version 2.01*

Tool Type: *Project Management Tool*

Description:

- *A tool useful for managing/scheduling technical projects*
- *Capable of producing Pert and Gantt charts, task lists, work breakdown summaries, project costs, schedules, etc.*
- *Suitable for all sizes of projects*

Environment:

- *IBM PC/XT/AT with minimum of 512K RAM*
- *Monochrome, EGA, CGA monitors*
- *Graphics plotter convenient for obtaining quality outputs*

Vendor Information:

*Software Publishing Corp. (408) 848-4391
1901 Landings Dr.
Mountainview, CA 94039*

Approximate Cost: \$350

User Comments, Pro:

Documentation is sufficient, but not overly complex. Very large projects can be managed with this software.

User Comments, Con:

Needs more control built into the generation of hardcopy (i.e., adjusting sizes). Takes a long time to recompute a schedule if a new task is added or deleted.

Additional Information: *Earl Creel, Organization 9132*

Tool Name: *Project Scheduler Network, Version 2.5*

Tool Type: *Project Planning and Management Tool*

Description:

- *Facilitates project scheduling and management by graphic means*
- *Implements Gantt charts, network diagrams using Critical Path Analysis Method (CPM), and Work Breakdown Structures (WBS)*
- *Interactive menu-driven user interface*
- *Quick "what-if" analysis functions for costs and resource allocation*
- *Report applications to supplement graphics*

Environment:

- *IBM PC or compatibles; Wang; TI Professional; HP150*
- *Minimum of 320K RAM required*
- *Supports IBM, Epson, HP Laserjet and Thinkjet, Toshiba, Okidata, and Xerox printers*
- *HP plotters (7470A, 7475A, 7550) recommended for graphics output*

Vendor Information:

*SCITOR Corp. (415) 570-7700
Commercial Products Division
250 Lincoln Centre Dr.
Foster City, CA 94404*

Estimated Cost: \$575

User Comments, Pro:

Color plot graphics are visually effective, easy to understand. Data entry is convenient. Compared to Harvard Total Manager, PSN is easier to use; reports and graphics can be interpreted more easily.

User Comments, Con:

Considerable manipulation may be required to get a satisfactory graphic layout of the network (precedence) diagram.

Additional Information: *Catherine Rosul, Organization 2812*

Appendix A

References

- BAB86 **Babich, Wayne**
Software Configuration Management: Coordination for Team Productivity, Addison-Wesley, 1986.
- BOE78 **Boehm, Barry**
Characteristics of Software Quality, TRW Series on Software Technology, North-Holland, 1978.
- BOE81 **Boehm, Barry**
Software Engineering Economics, Prentice-Hall, 1981.
- BRO75 **Brooks, Fred**
The Mythical Man-Month — Essays on Software Engineering, Addison-Wesley, 1975.
- CHA86 **Chao, Betty**
Design Guidelines for Human-Computer Dialogues, SAND86-0259, Sandia National Laboratories, May 1986.
- CHE77 **Chen, Peter**
The Entity-Relationship Approach to Logical Data Base Design, Q.E.D. Information Services, 1977.
- CHR81 **Christensen, K., G. Fitsos, and C. Smith**
"A Perspective on Software Science," *IBM Systems Journal*, Vol. 20, No. 4, 1981.
- COX86 **Cox, Brad J.**
Object-Oriented Programming; An Evolutionary Approach, Addison-Wesley, 1986.
- DEM79 **DeMarco, Tom**
Structured Analysis and System Specification, Yourdon Press, 1979.

- DEM82 **DeMarco, Tom**
Controlling Software Projects: Management, Measurement, and Estimation,
Yourdon Press, 1982.
- DEM87 **DeMarco, Tom**
Peopleware, Yourdon Press, 1987.
- DIJ72 **Dijkstra, Edsger**
A Discipline of Programming, Prentice Hall, 1973.
- DOD67 **Department of Defense**
Defense System Software Development, DoD-STD-2167A, February 29, 1988.
- FAG76 **Fagan, Michael E.**
"Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, Vol. 15, No. 3, 1976.
- FAG86 **Fagan, Michael E.**
"Advances in Software Inspections," *IEEE Trans. Software Engr.*, Vol. SE-12,
No. 7, pp. 744-751.
- FAI85 **Fairley, Richard E.**
Software Engineering Concepts, McGraw-Hill, 1985.
- FOW85 **Fowler, P.J., and A.F. Ackerman**
An Overview of Software Inspections, tape number GEN038, available from
Computing Education Center, Sandia National Laboratories.
- GAN79 **Gane, Chris, and Trish Sarson**
Structured Systems Analysis: Tools and Techniques, Prentice-Hall, 1979.
- GIL77 **Gilb, Tom**
Software Metrics, Winthrop Pub., 1977.
- GRA87 **Grady, R., and D. Caswell**
Software Metrics: Establishing a Company-Wide Program, Prentice-Hall, 1987.

- HAT87 **Hatley, D.J., and I. Pirbhai**
Strategies for Real-Time System Specification, Dorset House, 1987.
- HEH84 **Hehner, Eric**
The Logic of Programming, Prentice-Hall International, 1984.
- HIG77 **Higgins, David**
 "Structured Programming with Warnier-Orr Diagrams, Part I: Methodology,"
Byte, December, 1977.
- HIG78 **Higgins, David**
 "Structured Programming with Warnier-Orr Diagrams, Part II: Coding the
 Program," *Byte*, January, 1978.
- HUR84 **Hurley, Richard B.**
Decision Tables in Software Engineering, Van Nostrand-Reinhold, 1984.
- IEE83a **The Institute of Electrical and Electronic Engineers, Inc.**
IEEE Standard for Software Configuration Management Plans, ANSI/IEEE
 Std 828-1983.
- IEE83b **The Institute of Electrical and Electronic Engineers, Inc.**
IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE
 Std 729-1983.
- IEE84a **The Institute of Electrical and Electronic Engineers, Inc.**
IEEE Guide to Software Requirements Specifications, ANSI/IEEE Std 830-
 1984.
- IEE89 **The Institute of Electrical and Electronic Engineers, Inc.**
Software Engineering Standards, 3rd ed., IEEE, 1989.
- JAC83 **Jackson, Michael**
System Development, Prentice-Hall, 1983.
- JON86 **Jones, Capers**
Programming Productivity, McGraw-Hill, 1986.

- KER78 **Kernighan, B.W., and P.J. Plauger**
The Elements of Programming Style, 2nd ed., McGraw- Hill, 1978.
- LEM88 **Lemke, P.A., and M.J. Benson**
Software Development/Support Methodology, SAND88-2135, Sandia National
Laboratories, August 1988.
- MAR85 **Martin, James, and Carma McClure**
Action Diagrams: Clearly Structured Program Design, Prentice-Hall, 1985.
- MCC76 **McCabe, Thomas**
"A Complexity Measure," *IEEE Trans. Software Engr.*, Vol. SE-2, No. 4, 1976.
- MEL83 **Mellichamp, Duncan (ed.)**
Real-Time Computing with Applications to Data Acquisition and Control, Van
Nostrand Reinhold, 1983.
- MEY88 **Meyer, Bertrand**
Object-Oriented Software Construction, Prentice-Hall, 1988.
- MYE78 **Myers, Glenford**
"A Controlled Experiment in Program Testing and Code Walkthroughs/
Inspections," *Comm. of the ACM*, Sept. 1978, pp. 760-768.
- NIJ87 **Nijssen, G.M.**
"Knowledge Engineering, Conceptual Schemas, SQL, and Expert Systems: A
Unifying Point of View," Dept. Computer Science, Univ. of Queensland, St.
Lucia, Australia. Also available from NOVI (Dutch National Institute for
Informatics), Netherlands, 1986.
- PAG80 **Page-Jones, M.**
The Practical Guide to Structured Systems Design, Yourdon Press, 1980.

- PAS86 **Pascoe, Geoffrey A.**
"Elements of Object-Oriented Programming," *Byte*, Vol. 11, No. 8, pp. 139-144.
- PET81 **Peterson, James L.**
Petri Net Theory and the Modeling of Systems, Prentice-Hall, 1981.
- PRE87 **Pressman, Roger S.**
Software Engineering, A Practitioners Approach, 2nd ed., McGraw-Hill, 1987.
- SCH88 **Schroeder, D.H., and J.D. Mangum**
Process Guidelines for Sandia WR Software Development, SAND88-0024, Sandia National Laboratories, Albuquerque, NM, January 1988.
- SHE85 **Shefif, Y., E. Ng, and J. Steinbacher**
"Computer Software Quality Measurements and Metrics," *Micro. Reliab.*, Vol. 25, No. 6, pp. 1105-1150, 1985.
- SHL88 **Shlaer, Sally, and Stephen Mellor**
Object-Oriented Systems Analysis: Modeling the World in Data, Yourdon Press, 1988.
- SSGv1 **Sandia Software Guidelines**
Volume 1, *Software Quality Planning*, SAND85-2344, Sandia National Laboratories, Albuquerque, NM, Aug 1987.
- SSGv3 **Sandia Software Guidelines**
Volume 3, *Standards, Practices, and Conventions*, SAND85-2346, Sandia National Laboratories, Albuquerque, NM, Jul 1986.
- SSGv4 **Sandia Software Guidelines**
Volume 4, *Configuration Management*, SAND85-2347, Sandia National Laboratories, Albuquerque, NM, expected printing Feb 1990.
- TAU77a **Tausworthe, Robert C.**
Standardized Development of Computer Software: Part 1, Methods, Prentice-Hall, 1977.

- TAU77b **Tausworthe, Robert C.**
Standardized Development of Computer Software: Part 2, Standards, Prentice-Hall, 1977.
- TES86 **Tesler, Larry**
"Programming Experiences," *Byte*, Vol. 11, No. 8, pp. 195-206.
- VAL87 **Vallino, Jim**
"Tracking Code Modules," *PC Tech Magazine*, Vol. 5, No. 9, pp. 50-70.
- WAR78 **Warnier, J.D.**
Logical Construction of Programs, Van Nostrand Reinhold, 1978.
- WAR85a **Ward, P.T., and S.J. Mellor**
Structured Development for Real-Time Systems, Volume 1: "Introduction and Tools," Yourdon Press, 1985.
- WAR85b **Ward, P.T., and S.J. Mellor**
Structured Development for Real-Time Systems, Volume 2: "Essential Modeling Techniques," Yourdon Press, 1985.
- WAR86 **Ward, P.T., and S.J. Mellor**
Structured Development for Real-Time Systems, Volume 3: "Implementation Modeling Techniques," Yourdon Press, 1986.
- YOU79 **Yourdon, E., and L. Constantine**
Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, 1979.
- YOU85 **Yourdon, E.**
Structured Walkthroughs, Yourdon Press, 1985.
- YOU86 **Yourdon, E.**
Managing the Structured Techniques: Strategies for Software Development in the 1990's, 3rd ed., Yourdon Press, 1986.

Appendix B

Glossary and Acronyms

Where possible, definitions in this glossary are taken from the *IEEE Standard Glossary of Software Engineering Terminology*, [IEE83b]. They are included here to provide a single-source document for the reader.

- **analysis:** The examination of a topic to distinguish its component parts or elements separately or in their relation to one another.
- **ANSI:** American National Standards Institute.
- **audit:** An independent review for the purpose of assessing compliance with software requirements, specifications, baselines, standards, procedures, and instructions.
- **CASE:** Computer-aided **software engineering**.
- **CASE tool:** A computer-aided **software engineering** and development tool. [see **software tool**]
- **class:** A defined group of elements from an **object-oriented** system. The class definition provides the information required to construct and use **instances** of the class.
- **cohesion:** (1) A measure of the strength of association of the elements within a module. (2) The degree to which tasks performed by a single program module are functionally related. Contrast with **coupling**.
- **complexity:** The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics.
- **configuration management:** The process of identifying and defining the items in a system, and controlling the release and change of those items throughout the system life cycle.
- **context checking:** Given a software process, the action of testing the flow of information into and out of a model representing the process for coherency and consistency.
- **control flow diagram:** A graphical representation of a system showing the flow of control signals throughout the system.

- **coupling:** (1) A measure of the amount of information shared between two modules. (2) A measure of the interdependence among modules in a computer program. Contrast with **cohesion**.
- **data abstraction:** In **object-oriented** design, the process used to establish the data structure of each **object** so that it is only accessible by the object's own methods.
- **data dictionary:** (1) A collection of the names of all data items used in a software system, together with the relevant properties of those items. (2) A set of definitions of data flows, data elements, files, data bases, and processes referred to in a **data flow diagram**.
- **data flow diagram:** A graphical representation of a system, showing the logical flow of data and the processes transforming data, together with the sinks, sources, and stores for data.
- **data structure diagram:** A graphical representation of the ordering and accessibility relationships among items of data without regard to their actual storage configuration.
- **design:** The process of defining the software architecture, components, modules, interface, test approach, and data for a software system to satisfy specified requirements.
- **drawing system:** The formal system for defining Sandia-designed or controlled product and test equipment. This system includes standards for the content of engineering drawings, as well as methods for their creation, storage, duplication, and maintenance.
- **embedded computer system:** A computer system that is integral to a larger system whose primary purpose is not computational; for example, a computer system in a weapon, aircraft, command and control, or rapid transit system.
- **embedded software:** Software for an **embedded computer system**.
- **estimation:** (1) An appraisal or evaluation of the amount of time and resources required to develop software. (2) An evaluation or appraisal of the quality or value of an attribute, process, or product.
- **HIPO:** Hierarchy plus Input-Process-Output, a technique for representing the modules of a system as a hierarchy and for documenting each module.
- **IEEE:** The Institute of Electrical and Electronic Engineers, Inc.
- **information hiding:** An attribute of **object-oriented** systems in which information about the object is hidden within the module that represents the **object**.

- **inheritance:** A property of **object-oriented** systems which implies that all descriptive attributes of a class and the methods that operate on the **class** are inherited by subclasses of the class.
- **inspection:** A formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards and other problems. Contrast with **walk-through**.
- **instance:** An individual example of an **object** within an **object-oriented** system.
- **integrated tool:** A software tool that combines the capabilities of two or more development or maintenance functions in a single package.
- **life cycle:** [see **software life cycle**]
- **maintainability:** (1) The ease with which software can be maintained. (2) The ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements.
- **management:** The act of directing or managing a task.
- **metalanguage:** A language, usually with a rigid and formal syntax, in which another language may be defined.
- **method:** The operations **objects** are asked to perform on themselves. These operations are hidden within the module implementing the object.
- **methodology:** A body of methods, rules, techniques, and postulates employed in a field of study. [see **technique**]
- **metrics:** [see **software metric**]
- **milestone:** A scheduled event in a project used to measure progress.
- **mini-specs:** [see **process specification**]
- **model:** The representation of a real-world process, device, or concept.
- **modular:** The extent to which software is composed of discrete components such that a change to one component has minimal impact on other components.
- **object:** The focus of object-oriented techniques. Objects are abstractions of real-world items that are represented by data structures and accessed through program modules. [see **object-oriented**]

- **object-oriented:** A software development approach in which **objects** are the focus of the technique.
- **object-oriented design:** A disciplined approach to software design that focuses on the **object** as the basis for the design of modules.
- **p-specs:** [see **process specification**]
- **primitive process:** A process that has not been broken down or further divided into more processes.
- **process:** A unique, finite course of steps that leads to a particular result defined by the purpose or effect of the process.
- **process description:** [see **process specification**]
- **process specification:** An unambiguous, concise, textual description or definition of a **primitive process**.
- **programming language:** An artificial language designed to generate or express computer programs.
- **project management:** The administration of a project including organization, supervision, planning, scheduling, and implementation.
- **prototype:** (1) A functional form of a specified software design for evaluating requirements, design, and specification prior to full implementation of the software system. (2) A partial implementation of a system constructed to enable customers, users, or developers to learn more about a problem or a solution to that problem.
- **prototyping:** The application of a **prototype** in the process of software development.
- **quality assurance:** A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements.
- **rapid prototyping:** A prototyping technique that focuses on the development of the interface between the user and the computer. Rapid prototyping allows both the user and software developer to work together to facilitate the development of the **user interface**.
- **real-time:** Pertaining to the processing of data by a computer in connection with another process outside of the computer according to time requirements imposed by the outside process.
- **reliability:** The probability that software will not cause the failure of a system for a specified time under specified conditions.

- **requirement:** A condition or capability that must be met by a system or system component to satisfy a contract, specification, or other formally imposed document. The set of all requirements forms the basis for system development.
- **software:** Computer programs, procedures, rules, and associated documentation and data pertaining to the operation of a computer system.
- **software engineering:** The systematic approach to the specification, design, development, testing, operation, maintenance, and retirement of software.
- **software life cycle:** The sequence of events that begin when a software product is conceived and end when the software is no longer available for use.
- **software maintenance:** Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.
- **software metric:** A qualitative measure of the properties of software or of the processes used to produce software.
- **software quality:** (1) The totality of features and characteristics of a software product that bear on its ability to satisfy requirements specification. (2) The degree to which customers or users perceive that software meets their composite expectations.
- **software quality assurance:** [see **quality assurance**]
- **software tool:** A computer program used to help develop, test, analyze, or maintain another computer program or its documentation.
- **SRS:** Software requirements specification.
- **SSG:** Sandia Software Guidelines.
- **state transition diagram:** A graphical representation of the states of a system and the way transitions from one state to another occur.
- **states:** The conditions or modes of behavior of a system.
- **stand-alone tool:** A software tool that provides a single development or maintenance capability in one system.
- **structure chart:** A graphic chart for depicting the partitioning of a software system into modules, the hierarchy and organization of these modules, and the communication interfaces between the modules.

- **structured analysis:** A disciplined approach to requirements specification that stresses the development of a maintainable specification and communication with the user. The analysis is based on a technique using **data flow diagrams**, a **data dictionary**, and **process descriptions**.
- **structured design:** A disciplined approach to the design of software based on a hierarchical decomposition and implementation of system functions into software modules. Structured design has as one of its goals to maximize **cohesiveness** and minimize **coupling**.
- **structured methods:** Software development methods that are concerned with components of a system and the interrelationship between these components. Structured methods include **structured analysis**, **structured design**, and **structured programming**.
- **structured programming:** A well-defined software programming technique that normally incorporates strict use and implementation of structured program control constructs.
- **synchronization:** A process of maintaining one or more operations in step with each other.
- **technique:** A systematic means of achieving a desired goal. Contrast with **methodology**.
- **tool:** [see **software tool**]
- **user interface:** The means by which a user interacts with or communicates with a computer system. The interface involves both visual and behavioral actions and reactions of the user.
- **validation:** The process of evaluating software at the end of the software development process to ensure compliance with software requirements. [see **verification**]
- **verification:** The process of determining whether or not products of a given phase in the software development cycle fulfill the requirements established during the previous phase. [see **validation**]
- **walk-through:** A review process in which a designer or programmer leads one or more other members of the development team through a segment of design or code that he or she has written, while other members ask questions and make comments about technique, style, possible errors, violation of development standards, and other problems. Contrast with **inspection**.
- **waterfall model:** A model of the **software life cycle** in which the time periods or phases of the life cycle are arranged as the cascades in a waterfall. The software life cycle typically includes a requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and retirement phase.

Appendix C

The form enclosed in this appendix should be used to provide a description of CASE tools not listed in the volume or tools that have been upgraded. Descriptions of new or updated CASE tools will be saved and incorporated into future revisions of this volume. Completed forms should be sent to:

Editor, Sandia Software Guidelines
Organization 7254

SANDIA SOFTWARE GUIDELINES
Volume 5
Tool Description Form

Name/Org: _____ Phone: _____ Date: _____
 Tool Title: _____ Ver. No.: _____ (Beta Version? _____)
 Vendor/Developer _____

General System Functions:

Requirements Specs. Structured Design Structured Analysis
 Data Dictionary Testing Maintenance
 Prototyping Real-time Control Object Oriented Support
 Configuration Mgmt. Code Generation (lang: _____)
 Other _____

Additional Capabilities:

	Yes	No	Unsure	Comments
Multi-user security?				
Configuration management?				
Library creation by user?				
Macro definition by user?				
Suitable for all sized problems?				
Import/Export ASCII files?				

Environment:

	Yes	No	Unsure	Comments
Copy protected?				
Postscript supported?				
Mouse supported?				
Graphic card required?				

Supported on: PC _____ VAX _____ Other: _____
 Memory requirements: _____
 Which monitors are supported: _____
 Which printers are supported: _____
 What other equipment above a baseline system is required: _____

Purchase:

	Yes	No	Unsure	Comments
Site licenses available?				
Already at Sandia?				
Licensing problems?				
Maintenance contract available?				

Approx. total cost: _____

Support:

	Yes	No	Unsure	Comments
Is training available?				
Advisable in your opinion?				
Is there a hotline?				
If yes, is it useful?				

(OVER)

Ease of Use:

Is the product friendly to a user:

 New to software engineering?

 Familiar with software engineering?

Is there on-line help?

Is the report writing capability adequate for producing a hardcopy set of documents?

Yes	No	Unsure	Comment

Comments:

Tool Description (brief description):

User Comments:

 Pro:

 Con:

 Name

 Org.

 Phone

SNL Contacts:

References: [author, title, date, publication]

Please attach one or two pages of sample output if possible.
Return form to: Lynn Ritchie, SNLA/Org. 7254

INDEX

A

Ada, 14, 29, 37, 46, 47, 64
ADADL, 64
AGT, 37, 64, 65
AIDA, 38
analysis
 structured, 81
ANALYZ, 58
ANSI, 76
arc, 29
audit, 27, 56, 76

B

Backus-Naur form, 8
Boehm, 17, 70
Brooks, 20, 60

C

C, iii, 3, 14, 29, 32, 46, 64,
 65, 66, 70, 74, 75, 82
CASE, iii, 2, 4, 7, 12, 15,
 21, 22, 23, 24, 25, 26,
 27, 28, 29, 30, 32, 33,
 36, 40, 57, 67, 76, 82
CASE tools, iii, 2, 4, 7, 12,
 15, 21, 22, 24, 25, 26,
 27, 28, 29, 30, 32, 33,
 36, 57, 67, 82
Caswell, 17, 71
Chen, 9, 70
class, 5, 9, 13, 14, 15, 76,
 78
CMS, 52
code frame, 29
code generation, 22, 29, 46
cohesion, 10, 76, 77
command-line, 26
complexity
 metric, 62
computer-aided software
 engineering, 21, 76

configuration management,
 i, iii, 18, 19, 22, 25, 30,
 32, 51, 52, 53, 54, 55,
 56, 70, 72, 74, 76
Constantine, 11, 12, 75
context checking, 22, 24,
 25, 28, 29, 40, 76
control flow diagrams, 15, 40
control flows, 15
control specifications, 15
control stores, 15
copy protection, 22, 23
coupling, 10, 76, 77, 81

D

data abstraction, 77
data base, 9, 25, 27, 28, 29,
 41, 42, 46, 47, 49, 70
data dictionary, 8, 22, 25,
 28, 37, 39, 40, 41, 43,
 49, 50, 77, 81
data flow diagram, iv, 8,
 15, 28, 29, 77
Data Item Descriptions, 6
data structure diagram, 77
data structures, 11, 13, 16,
 28, 76, 78
debugger, 30
DEC, 47, 64
DeMarco, 5, 17, 20, 25, 44,
 50, 70, 71
DEMO, 34
design, iii, 3, 7, 10, 11, 12,
 13, 14, 15, 16, 21, 24,
 25, 27, 31, 32, 35, 36,
 37, 38, 39, 40, 41, 43,
 45, 46, 47, 48, 49, 63,
 64, 70, 73, 75, 77, 78,
 79, 80, 81
DIDs, 6
Dijkstra, 12, 71
documentation, i, 1, 5, 6,
 18, 21, 24, 25, 26, 27,
 37, 38, 42, 43, 45, 47,
 48, 51, 52, 53, 59, 61,
 66, 68, 80

DoD-STD-2167A, 6, 71
drawing definition system, 19
drawing system, 77

E

embedded
 computer system, 77
 software, 77
estimation, 16, 17, 71, 77

F

Fagan, 18, 71
Forth, 14
FORTRAN, 29, 58, 59, 61,
 66
FORTRAN-LINT, 59

G

Gane, 7, 71
Ganitt, 30, 68, 69
Gilb, 17, 71
Grady, 17, 71
GRED, 64, 65

H

Halstead, 16
Harvard Total Program
 Manager, 68
Hatley, 15, 72

I

icon, 26
IDMS, 41
IEEE, i, 3, 6, 19, 20, 71,
 72, 73, 76, 77
information hiding, 77
information modelling, iii,
 9, 10, 14, 36, 42
inheritance, 78
inspection, 18, 78, 81

inspections, iii, 17, 18, 71
instance, 9, 17, 25, 28, 78
integrated, iii, 22, 25, 27,
 28, 30, 36, 39, 41, 46,
 47, 78
integrated tool, 22, 28, 78
integrated tools, iii, 25, 28, 30

J

Jackson, 12, 72
JCLCHECK, 60
Jones, 17, 72

K

Kernighan, 12, 73

L

Language Sensitive Editor,
 29
level checking, 28
Librarian, 54
libraries, 13, 22
library, 3, 38, 39, 42, 46,
 52, 53, 54
license, 23, 40, 45, 47, 52,
 53, 59, 60, 61
licensing, 22, 23
LSE, 30

M

macro, 38, 39, 42, 46
maintainability, 10, 21, 62,
 78
maintenance, 2, 4, 5, 13,
 16, 17, 18, 23, 25, 30,
 31, 41, 43, 45, 54, 57,
 59, 60, 61, 63, 64, 77,
 78, 80, 81
McCabe, 16, 62, 73
Mellor, 10, 14, 15, 74, 75
menu, 25, 26, 27, 34
menus, 24, 26, 50

messages, 13, 61
method, 2, 4, 5, 7, 8, 9, 10,
11, 12, 14, 15, 23, 26,
30, 40, 69, 78
methodologies, 1, i, iii, 1, 2,
4, 5, 15, 17, 20, 21, 36
methodology, i, 2, 5, 7, 18,
21, 23, 40, 42, 44, 45,
47, 48, 49, 50, 73, 78, 81
metric, 16, 62, 80
metrics, iii, 16, 17, 57, 71,
78
milestone, 78
MIRAGE, 35
MMS, 53
model, 4, 7, 9, 13, 15, 17,
24, 47, 76, 78, 81
Modula-2, 66
modular, 10, 78
modularity, 10, 13
module specification, 11
mouse, 23, 26, 27, 35, 39,
44, 50

N

NIAM, 9, 42

O

object, 13, 29, 77, 78, 79
object-oriented, iii, 10, 11,
13, 14, 37, 49, 70, 73,
74, 76, 77, 78, 79

P

p-specs, 8, 79
Page-Jones, 11, 73
Pascal, 14, 29, 46, 66
PC-IAST, 42
PCSA, 50
PERT, 30, 68
pic, 27
Pirbhai, 15, 72

Plauger, 12, 73
PostScript, 27, 37, 50, 64, 65
primitive, 8, 79
primitive process, 79
procedure-oriented, 11, 13, 14
process, 4, 5, 8, 13, 14, 15,
16, 17, 18, 20, 21, 25,
27, 28, 31, 40, 50, 74,
76, 77, 78, 79, 81
process specification, 8, 40, 79
programming language, 2,
14, 79
project management, iii, 16,
19, 20, 30, 31, 32, 67,
68, 79
project management plan,
20
Project Scheduler Network,
69
projection, 17
Prokit, 43
prototype, 6, 7, 35, 47, 64, 79

Q

query language, 27, 28

R

rapid prototyping, 7, 79
real-time, iii, 9, 14, 15, 16,
25, 36, 38, 40, 47, 49,
72, 73, 75, 79
real-time systems
analysis, 15, 49
redraw, 24, 26, 28
reliability, 16, 17, 21, 79
report generation, 21, 27, 60
requirement, 6, 18, 80
reverse engineering, iii, 31,
32, 37, 63, 64, 66

S

SA Tools, 44

Sandia Software Guidelines,
 i, 1, 2, 3, 19, 74, 80, 82
 Sarson, 7, 71
 SDM, 45
 Shefif, 17, 74
 Shlaer, 10, 14, 74
 Smalltalk, 14
 software
 configuration
 management, iii, 18, 19,
 32, 51, 70, 72
 design, 46, 79
 inspections, iii, 17, 18,
 71
 life cycle, i, iv, 4, 5, 25,
 32, 80, 81
 maintenance, 80
 metric, 16, 80
 metrics, iii, 16, 17, 57,
 71
 quality, i, ii, 3, 16, 17,
 70, 74, 80
 quality assurance, i, ii,
 80
 tool, iii, 2, 25, 64, 78,
 80
 software engineering, 2, 3,
 5, 12, 15, 18, 20, 21, 28,
 38, 43, 70, 71, 72, 74,
 76, 80
 Software Quality Assurance
 Division, ii
 Software Through Pictures,
 46
 source code, 16, 17, 19, 29,
 31, 54, 55, 57, 63, 65,
 66
 Source Tools, 56
 SRS, 80
 SSG, 80
 stand-alone, iii, 25, 28, 30,
 36, 80

stand-alone tool, 80
 standard, i, 5, 6, 19, 20, 22,
 23, 72, 76
 standards, i, 3, 18, 27, 60,
 72, 74, 75, 76, 77, 78, 81
 state transition diagram, 80
 STATEMATE, 47, 48
 states, 9, 12, 15, 80
 static analysis, 57, 58, 59,
 61
 static code analysis, iii, 32,
 57
 structure chart, iv, 11, 31,
 80
 structured
 analysis, iii, 6, 7, 9, 10,
 11, 15, 25, 36, 39, 40,
 41, 43, 44, 45, 46, 49,
 50, 70, 81
 design, iii, 10, 11, 12,
 16, 25, 75, 81
 methods, 12, 13, 81
 programming, 12, 81
 symbolic debugger, 30
 synchronization, 14, 15, 81
 S_PAT/CAM, 62
T
 Teamwork, 49, 50
 technique, 5, 6, 7, 9, 10, 12,
 18, 27, 30, 42, 77, 78,
 79, 81
 testing, 4, 5, 10, 17, 20, 28,
 38, 45, 47, 73, 76, 80
 tool description form, iii
 tools, 1, i, iii, 1, 2, 4, 7, 12,
 15, 19, 21, 22, 24, 25,
 26, 27, 28, 29, 30, 31,
 32, 33, 36, 38, 40, 44,
 46, 48, 51, 56, 57, 63,
 67, 71, 75, 82
 Tree Diagrammer, 66
 truth tables, 8

U

user interface, iii, 7, 24, 25,
26, 32, 33, 34, 35, 54,
69, 79, 81
user-friendly, 24, 27, 38,
39, 42, 46, 52, 53

V

validation, 81
verification, 27, 41, 81
version control, 19, 22, 25,
30, 55

W

walk-through, 81
Ward, 15, 75

Warnier, 12, 75

Warnier-Orr diagrams, 12,
72

waterfall, 17, 24, 81

windowing, 24, 25, 26, 28,
30

windows, 25, 26, 34

WR, 2, 20, 74

Y

Yourdon, 11, 12, 18, 20, 25,
39, 70, 71, 73, 74, 75

DISTRIBUTION:

1	Martin Marietta Energy Systems, Inc.	1	2336	S. C. Richards
	Attn: Melissa Z. Smith, Y-12	1	2364	J. F. Jones, Jr.
	Vice-Chair, SQA Subcommittee	1	2600	L. D. Bertholf
	Y-12 Plant	1	2600A	P. A. Lemke
	PO Box 2009	1	2601	J. C. Garrison
	Building 9723-11A, Mail Stop 8127	1	2610	D. C. Jones
	Oak Ridge, TN 37931	1	2614	A. R. Iacoletti
		2	2614	A. A. Elsbernd
1	Allied-Signal Aerospace	250	2614	B. L. Straba
	Kansas City Division, D/418	1	2620	K. O. Waibel
	Attn: John Vic Grice, KCD	1	2624	T. L. Ferguson
	Secy, SQA Subcommittee	2	2624	J. R. Schofield
	PO Box 419159	2	2625	D. S. Campbell
	Kansas City, MO 64141-6159	1	2634	S. K. Fletcher
		1	2634	C. K. Haaker
1	Allied-Signal Aerospace	2	2634	S. L. Jean-Louis
	Kansas City Division, D/418	1	2640	E. J. Theriot
	Attn: Don Schilling, KCD	1	2800	W. E. Alzheimer
	JOWOG-39 Co-Chairman	1	2810	D. W. Doak
	PO Box 419159	1	2812	J. A. Wisniewski
	Kansas City, MO 64141-6159	2	2812	A. C. Beradino
1	131 R. H. McGee	2	2812	C. C. Rosul
2	131 R. N. Banwart	2	2812	S. L. K. Rountree
1	151 N. A. McEwen	1	2814	P. A. Erickson
2	151 R. T. Gauna	5	2814	M. E. Olson
1	400 H. D. Pruett	1	2820	G. Carli
1	1265 P. L. McAllister	2	2821	K. W. Osburn
1	1410 P. J. Eicker	2	2825	J. K. Sharp
1	1412 G. S. Davidson	1	2825	O. H. Bray
1	1414 R. C. Lennox	2	2825	N. H. Stevens
1	1424 M. P. Sears	1	2834	D. L. Janni
1	1531 S. L. Thompson	1	2854	R. E. Thompson
1	1531 J. M. McGlaun	1	2854	S. C. Babb
1	1551 H. R. Spahr	2	2854	D. P. Patrick
1	1555 G. F. Polansky	1	3411	L. A. Malczynski
10	1556 W. L. Oberkampf	1	5100	H. W. Schmitt
1	2300 J. L. Wirth	1	5145	D. H. Schroeder
1	2310 M. K. Parsons	2	5145	M. W. Sharp
1	2311 T. D. Donham	1	5173	R. F. Davis
2	2311 C. E. Nuckolls	2	5173	D. L. Harris
1	2312 D. J. Allen	2	5173	B. N. Malm
2	2312 J. D. Mangum	1	5173	A. L. Yates
1	2314 D. M. Small	1	5221	D. D. Spencer
1	2315 M. J. Smartt	2	5246	L. M. Desonier
2	2315 B. P. Chao	1	5246	R. P. Syler
1	2330 J. H. Stichman	1	5268	M. S. Allen
2	2336 N. R. Kolb	2	5268	M. R. Moseley
1	2336 E. J. Nava			

DISTRIBUTION: (continued)

1	5268	S. J. Weissman	1	8233	J. M. Harris
1	6447	D. A. Brosseau	1	8233	R. Y. Lee
1	6452	G. C. Giesler	1	9127	C. E. Olson
1	7111	L. R. Hill	1	9131	D. D. Boozer
1	7131	L. F. Brady	1	9131	J. A. Hollowell
1	7200	C. H. Mauney	2	9132	E. E. Creel
1	7233	R. E. Smith	1	9211	G. J. Dodrill
1	7250	G. T. Merren	1	9215	F. N. Hill
1	7252	D. W. Bushmire	1	9215	K. G. Weber
2	7252	G. D. Bowling	1	9220	G. H. Mauth
1	7252	M. E. Prickett	1	9221	L. M. Grady
1	7253	G. E. Dahms	2	9221	J. P. Kern
10	7254	M. A. Blackledge	1	9221	J. L. Williams
5	7254	K. E. De Jong	1	9224	L. J. Ellis
5	7254	J. P. Franklin	2	9224	K. M. Erickson
10	7254	L. T. Ritchie	2	9224	A. M. Maxted
5	7254	E. H. Tomlin	2	9224	R. W. Simons
5	7254	S. L. Trauth	1	9243	W. R. Pfarner
1	7262	R. B. Ronan	2	8524	J. A. Wackerly
1	7263	G. W. Mayes	5	3141	S. A. Landenberger
1	8134	M. H. Rogers	8	3141-1	C. L. Ward for DOE/OSTI
1	8134	K. A. True	3	3151	W. I. Klein
1	8230	W. D. Wilson			
1	8233	F. J. Cupps			

Second Printing, September 1992

326 M. A. Blackledge (500)