1988

Ada as a design specification language

Barbara Brunner Gibson

Rochester Institute of Technology
School of Computer Science and Technology

ADA AS A DESIGN SPECIFICATION LANGUAGE

by
Barbara Brunner Gibson

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by: Professor Peter G. Anderson

Professor Rayno Niemi

Professor Jeffrey A. Lasky

May 3, 1988

ADA AS A DESIGN SPECIFICATION LANGUAGE

I, Barbara B. Gibson hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

May 9, 1988
Date

ABSTRACT

The primary thesis objective is research into current approaches to design specification languages, emphasizing Ada. Requirements specification is touched upon. Design specification is explored and related to requirements and implementation. The role of language in design is discussed, as well as objectives of the design specification and features that a specification language should provide in order to meet those objectives. Formal language is contrasted with natural language. Some formal specification languages are described, both Ada related and not Ada related.

The secondary objective, the thesis project, is to illustrate a design specification in a formal language, Ada. The purpose of the project is to compare the Ada expression of an example design with the natural language specification for the same system.

KEY WORDS AND PHRASES

Ada, Design, Programming Languages, Requirements, Software Engineering, Specification.

COMPUTING REVIEW SUBJECT CODES

D.2.1. Requirements/Specifications - languages, methodologies, tools.

D.2.10. Design - methodologies, representation.

CONTENTS

CONTENTS

CONTENTS

ADA* AS A DESIGN SPECIFICATION LANGUAGE

SECTION 1. INTRODUCTION AND BACKGROUND

1.1. THESIS OBJECTIVES

The chief objective of this paper is to explore formal software system specification languages, with emphasis upon Ada. The secondary objective is to compare a natural language design for an example software system with the design for the same system specified in Ada.

The project life cycle will be discussed briefly as background for later treatment of both requirements and design specification. I will describe objectives of the design specification followed by objectives of the language for expressing the design specification. I will contrast formal and natural language for documenting system design and then describe some formal design languages, both Ada related and not related to Ada. Later sections of the thesis will describe the thesis project and compare the natural language and the Ada designs for the example system. Examples of the two designs are provided in the appendices. The last section of the paper will discuss problems in using formal languages for specification and also another approach that could be taken to assessing Ada as a specification language. Finally, I will draw conclusions about using formal languages for design specification and about using Ada in particular.

1.2. GENERAL DESCRIPTION OF THE PROBLEM AREA

Progress and growth in computer hardware has far exceeded that of software. Software advances have been made mostly in programming languages; productivity has risen slowly. Software development is expensive and unpredictable both economically and in terms of the correctness and efficiency of the product. To reduce the overall cost of software development, strategies are needed that go beyond programming techniques.

---

* "Ada" is a registered trademark of the U.S. Government, Ada Joint Program Office.

Attention currently focuses on non-programming phases of the life cycle: requirements and design specification. "Software design and development is the weakest link in the system development process." [1] "Up to sixty percent of software errors can be traced back to design." [2] Methodologies for requirements and design have been developed in an attempt to control and direct the development process. A methodology or life cycle model "is an effective way to think, communicate thoughts, and to practically realize these thoughts . . . [It] can make the difference between understanding a system and not understanding it." [3]

Because specification and design activities affect each phase of the software life cycle, a brief review of the software life cycle is appropriate.

1.2.1. THE SOFTWARE LIFE CYCLE

The traditional, or "waterfall," life cycle model originated with W.W. Royce in 1970. [4] In this model, a system passes sequentially through a series of phases, from the development phase to delivery, which initiates the operation and maintenance phase, which continues until the system is retired. The development phase is subdivided into three phases, each of which terminates when the client approves the product of the phase. (1) The requirements analysis phase produces the requirements specification of system functions and resources. This is a preliminary, very high level design. (2) The design phase produces the system design specification. This is the functional design or preliminary product specification. (3) The implementation phase produces the implemented system. Implementation consists in coding, testing, and integrating program units to produce an integrated system.

The required effort varies with each phase. Coding and unit testing accounts for the smallest percentage of effort, which is why advances in programming languages alone can not produce sufficient reductions in software costs. The Department of Defense (DOD) assigns approximate software costs as follows: 70% maintenance and modification; 15% testing and integration, 4.5% coding and debugging; 10.5% requirements and design. [5]

69% Of the errors requiring maintenance can be traced to the requirements or design phase. The high cost of maintenance is due to problems arising from insufficient and incomplete documentation, inconsistency between documentation

and code, design that is difficult to modify and test, and insufficient records of past maintenance. The cost of correcting software errors in the operational phase is 64 times the cost of correcting in the detailed design phase and 90 times the cost of correcting in the preliminary design (requirements) phase. The cost of correcting in the validation phase is almost 13 times design and 18 times greater than requirements. [6] The conclusion from these figures is that (1) improvements in requirements and design specification will decrease maintenance and also facilitate whatever maintenance is necessary; (2) the cost benefit of decreased effort in the later life cycle phases of testing and maintenance greatly outweighs the cost of increased effort in the early phases of the life cycle.

The waterfall model is inadequate as a model for software development because frequent overlaps of phases are needed. In the model, the phases are sequential. In practice, overlapping occurs when information surfaces in a later phase that necessitates revision of work done in a previous phase or phases.

Hamilton and Zeldin point out additional shortcomings of the traditional life cycle model: [7] (1) The manual processes during and between all phases of development encourage the introduction of new errors. (2) The target system is described in a different language for each different phase of development; correspondence must be proven for each translation from the previous phase. (3) The model is inefficient in its ordering of processes; the majority of validation and verification is at the end; errors remain in the system too long and encourage new errors. (4) Systems are not functionally understood; everything must be treated as an unknown until the system is complete because dynamic verification must always be performed on all the objects within the system. (5) There are no fixed semantics; the semantics changes with the arrival of new languages with different syntax. (6) Requirements may contain implementation details which depend upon the implementation language or the hardware environment.

An alternative to the waterfall model is to view system development as a gradual concretization of abstractions. Weber and Ehrig [8] assert that system development should be a gradual refinement of modules. Modules are self contained software units that encapsulate data types and operations, providing simultaneous decomposition of data and operations. There should be no operations and global data that are not part of a module. During the entire development process, developers should use a single language that becomes more formal as modules become more completely realized.

The choice of life cycle model determines how tasks are accomplished during the development process, and how the products of development phases are expressed. These development products are also models - models for the system in progressive states. Models are the most important products of requirements and design. They affect every activity of the life cycle.

1.2.2. EFFECT OF NEW REQUIREMENTS METHODS ON THE LIFE CYCLE

New methods for requirements specification affect the traditional life cycle approach. Any method that measurably improves the quality of requirements definition has the effect of decreasing the effort required for later life cycle steps. Methods such as prototyping or automatic code generation radically change the life cycle. When formal language or graphic expression are introduced into the requirements phase, the distinction between requirements and design becomes blurred.

Prototyping leads to a gradual clarification and refinement of requirements specifications. Simulation of requirements through the prototype predicts the consequences of implementation without actual implementation of the completed system. The prototype may become the basis for the implementation; some of the implementation activity is thus transferred to the requirements phase of the life cycle. The effect of prototyping is a higher quality design that results in decreased effort in the implementation and maintenance phases.

Scenario-based prototyping is described by Hsia, Young, and Jiam. [9] To avoid losing sight of the system's objectives, they recommend viewing the system from the external, user's perspective. The system should be partitioned by utility, without attention to details of construction as is the case when the system is viewed from the internal, designer's perspective.

A scenario is a "system sketch," a series of screens which accomplish the user's objectives and serves to clarify the user's requirements. The developer selects representative scenarios and groups them into clusters according to system use. These clusters then form the basis for subsystems which can be developed incrementally.

Balzer, Cheatham, and Green describe "a new paradigm" for software development based on automation. [10] Stages in the new life cycle are:

1. Formal specifications.

2. Generation of a prototype from the formal specifications - the prototype becomes the new specification.
3. Validation of the prototype against the intent.
4. Implementation, machine aided, from the prototype. This is really optimization of the prototype.
5. Testing is eliminated.
6. Maintenance is performed on the formal specification.

The automated paradigm eliminates the possibility of undocumented development. It also results in reusable specifications.

Requirements may be expressed graphically rather than linguistically. The graphic requirements expression may then be automatically translated into a formal language and implemented. The extent to which a graphic requirements system is useful depends upon what concepts are available in the graphics vocabulary and the human factors of the graphics system interface.

Specification of functionality in the requirements phase may be replaced by specification of an evaluation process and a set of acceptance criteria. This is especially useful when the problem is poorly specified, algorithms are ill-defined, or when there is no algorithm that fits every possible case. A medical expert system for diagnosis of a class of diseases would be an appropriate use for specifying requirements in this way. For example, the diagnosis would be required to be in 90% agreement with a group of experts on a predefined set of cases. [11]

Knowledge based systems may assist in requirements definition. According to Gruia-Catalin Roman [12], the topic is now being discussed at software engineering conferences.

1.2.3. EFFECT OF NEW DESIGN METHODS ON THE LIFE CYCLE

Formalized design specification can radically alter the software life cycle. Automatic generation of code from a formal language design eliminates testing for correctness of the code. Only user acceptance testing is required. In addition, automatic code generation permits maintenance upon the specification rather than the implementation. This improves the ease and correctness of maintenance.

A significant number of formal methods for design specification have graphical interfaces. The designer can enter a graphical model of the system under construction, expressing data structures, data flow and control flow. The

graph model is then translated into graph language. The graph language permits graphs for each phase of the system to be expressed in a compatible manner and to be related through graph rewriting rules. This provides a method of verifying the validity of the forward transformation from the previous phase.

1.2.4. EFFECT OF SINGLE LANGUAGE METHODS ON THE LIFE CYCLE

Using a single language for design and implementation improves the quality of the implementation because nothing is omitted or modified in translating from design to implementation. It facilitates implementation because the implementation is a refinement or natural extension of the design. The design is expressed in terms of structures that can be directly realized in the implementation language.

A single language permits an incremental or semi-incremental life cycle. In the incremental life cycle, design is merged into implementation. Implementation is a series of steps, each of which gradually adds to the existing system by realizing a part of the system that had previously existed only in intention. In the semi-incremental life cycle, design and implementation overlap because the design can be easily converted to code and tested at any design stage.

Rajlich discusses 3 common paradigms for design and implementation using Ada as the single language. [13]

(1) Bottom-up incremental programming is forgiving of wrong decisions, but may result in a system that does not meet its specifications. Some, notably Nicholas Wirth, reject the bottom-up method for this reason.

(2) Top-down semi-incremental programming is natural and easy to use, but limits the possibility of parallel efforts on a project (because the design is incomplete). There are three steps to this method: decomposition and completion of variables and procedures; abstraction - replacing variables by types and combining procedures into single, more abstract procedures; definition - defining unfinished entities in terms of the programming language primitives.

(3) Large-small traditional programming permits high parallelism, but requires a detailed, documented design before implementation begins. In this method, the module specifications are defined first and the bodies developed later. Errors in module specifications thus have a ripple effect in other modules.

SECTION 2. REQUIREMENTS SPECIFICATION

The requirements specification describes system functionality. "Requirements definition is founded on showing what the functional architecture is, also showing why it is what it is, and constraining how the system architecture is to realize it in more concrete form." [1]

The quality and completeness of the requirements documentation determine the success of the system. During the requirements phase, developers must achieve a precise understanding of the data. System functions can be identified from the flow and definition of the data. [2]

2.1. THE REQUIREMENTS PROCESS

Requirements specifications are developed from a thorough understanding and description of the problem and what is needed for a solution. Developing requirements includes three phases, each with a different focus for analysis. [3]

First, in needs analysis, developers and client work together to define problems and what is needed to solve the problems. They determine which problems to solve first and then clarify objectives for the system. It is important to separate what must be done from how it is to be done, and to describe the whole problem, the overall structure of activities from both horizontal and vertical viewpoints.

Second, in functional requirements analysis, they define the functions to realize the system objectives. The client's active participation in this phase is critical so that the system is analyzed from the viewpoint of how it is used rather than how it is to be constructed.

The third phase in the requirements process is the operational requirements analysis. The activity here is to assess the resources and materials required for each function in order to realize the objectives. Estimates of the cost to develop, maintain and operate the system are compared to the projected benefits in order to determine feasibility and justification.

Requirements analysis is an iterative process of analyzing the problem, documenting insights into the requirements, and checking the understanding that

was gained. [4] Checking the understanding involves reviewing the current state of the requirements in light of new insight, performing additional analysis where appropriate, and revising the requirements as necessary. The problem statement must be syntactically accurate, internally consistent, and as complete as the current understanding permits. Analysts and clients cooperatively ensure that the documented requirements represent the problem accurately and ensure that what is represented is really what is desired.

The analysis process requires an environment that includes a methodology appropriate to the problem, tools to gather information and to describe models, organization to permit easy access to facts, and support for communication in presentation and discussion. [5]

2.1.1. HITACHI GROUP METHODOLOGY

A methodology is required in order to analyze large scale information systems where the client's requirements are vague and relations cross operational sections. It is difficult to recognize the activities that are related to plural sections of an organization. A methodology is used to organize data, to extract the problems and needs, and to develop the requirements analysis. It provides organization in order to visualize the relationships among the elements concerned with each phase of the analysis. An example is the Hitachi group method, which uses a tool, PPDS (Planning Procedure to Develop Systems). [6] This method uses three basic models: objectives trees as an aid in needs analysis; functional activity flow as an aid in the functional requirements analysis; and operational activity flow as an aid in the operational requirements analysis.
Other matrices, graphs and trees are developed in the process of producing the basic models and relating them. PPDS assists in the construction of the objectives trees. It provides graphic displays for visualizing and maintaining the needs documentation based on incomplete understanding of the requirements.

2.1.2. SADT METHODOLOGY

SADT (Structured Analysis and Design Technique, SOFTECH) is perhaps the best known methodology for describing system requirements. This modeling technique may also be used for system design, project management, and other

applications. [7] The SADT methodology provides a disciplined approach to analysis which helps the analyst to work out a clear understanding of the subject. Complex subjects are progressively partitioned hierarchically into 6 or fewer "chunks" that are easy to grasp. Three fundamental questions of analysis must be answered for each subject: why, what, how. The process of answering these questions provides context analysis (why), functional specification (what), and design constraints or boundary conditions (how). "Why some feature is needed molds what it has to be, which in turn molds how it is to be achieved." [8]

SADT uses natural language in blueprint-like graphic diagrams to show necessary relationships. Each topic must be carefully delineated so that the reader can grasp the whole message. The box and arrow conventions of the data models and activity models structure the meaning of the words to remove ambiguity.

2.1.3. FORMALIZATION OF REQUIREMENTS

The trend is toward formalization of requirements specifications; e.g., SREM (see section 7). Formal specification is an intermediate step between natural language requirements specification and design. Natural language is used to express the problem, needs, and objectives: the end-user requirements documentation. Formal language is used for technical documents which provide specifications for computer professionals. The formal specification complements the natural language specification and can aid in improving it. Graphics tools can be used to translate formal elements into a representation that is more easily understood. [9]

Formal foundation is one of several criteria for classifying requirements specification techniques. These classification criteria, listed by Gruia-Catalin Roman [10], suggest the variety of types of methodology for analyzing requirements. Roman classifies the techniques according to:

1. Formal foundation: the theory forming the basis for the technique, e.g., data flow, use of finite-state machines, stimulus-response paths, communicating concurrent processes, functional composition, and data-oriented models.
2. Scope: the type of requirements the technique attempts to express (functional and non-functional).

3. Level of formality: the extent to which the technique is machine processable.
4. Degree of specialization: the size of the class of problems for which the technique is applicable.
5. Specialization area: the class of problems for which the technique is applicable.
6. Development method: the approach used to construct the specification, e.g., prototyping, or mixed prototyping and traditional.

2.1.4. AUTOMATION OF REQUIREMENTS

Specialization of a requirements technique increases the potential for automation. Limiting the technique to a particular class of application makes the technique more easily analyzed. This also increases the technique's potential for representation by graphic methods, which makes it easier to use. [11]

Requirements may be automated by specification in a formal language and generation of a prototype or by graphic specifications and generation of code. Another method of automation is through rule based systems. [12] These permit the user to state a set of decision rules for a problem. The system then produces a set of actions that follow the rules.

2.2. THE REQUIREMENTS SPECIFICATION

The end-product of requirements analysis is the requirements specification, which describes the general constitution (objectives) of the product, independent of any realization. The requirements document should include the functional specifications, constraints, and context. This becomes the basis for the subsequent system design.

2.2.1. FUNCTIONAL REQUIREMENTS SPECIFICATION

The specification of functional requirements should list all functions, both manual and automated. The functional requirements provide a conceptual model of the states and behavior of the system and its environment. These requirements must be expressed in terms of objectives to be met and must not contain elements

that affect how the system will be constructed. The document should include the syntax and semantics of all input. It should describe input, operations, and the required results in terms of operations. This includes a description of error processing.

It is important to document the environment for each function. Constraining the environment can reduce system complexity and thus decrease effort required for implementation and maintenance.

Morton and Freburger [13] advise that the functional specifications should include a controlling philosophy for the system. The document should state whether the system should be screen based, menu based, or dialogue based. It should specify the attitude toward default parameters. And it should state whether ease of use means consistent or intuitive for each specific case.

2.2.2. NON-FUNCTIONAL REQUIREMENTS

The non-functional requirements specification documents the system constraints and their effect upon the system behavior. Constraints include the user interface, performance, reliability, safety, and cost. Performance includes requirements for time and space, reliability, security. Survivability is specified as a performance constraint for defense systems or for disaster planning purposes (survival of system elements). [14] Operating constraints document requirements for operating personnel such as skill level and location. Life cycle constraints may describe: design qualities such as maintainability or compatibility; limits on the development process such as deadlines, resource availability, programming standards. The document also specifies policy and legal constraints.

2.2.3. REQUIREMENTS CONTEXT

Context analysis should provide background for the requirements; requirements alone are insufficient for understanding the client's needs. The developer must model the environment to fully understand the problem and needs which determine the system objectives and characteristics. The specification should describe the organization and where the system will fit, showing how it meets the needs.

2.2.4. REQUIREMENTS GUIDELINES

The requirements must be complete, consistent, testable, traceable, feasible, and flexible. [15] Barry Boehm defines these terms. [16] Completeness requires all parts of the specification to be present and each part to be fully developed. Consistency means that the specification's provisions do not conflict with each other (internal consistency) or with governing entities, specifications and objectives (external consistency). To be testable, the specification must be specific, unambiguous and quantitative wherever possible. This enables techniques for determining if the software satisfies the specification. Traceability requires all the items in the specification to have clear antecedents in earlier specifications or statements of the system objectives. For feasible requirements, the life-cycle benefits of the system must exceed the cost. The system must therefore be maintainable, reliable and "human-engineered." Feasibility implies identifying and resolving technical, cost-schedule, and environmental issues of risk before committing to system development.

Gruia-Catalin Roman describes additional properties of an effective requirements specification. [17]

1. Appropriateness - captures the concepts of the component's role in the environment.
2. Conceptual cleanness - is simple, clear, understandable. This may be sacrificed for efficiency if the requirements are used by tools alone.
3. Constructability - manifests a systematic approach, a structuring that separates concerns and gives ease of access to frequently used information.
4. Precision - completeness, consistency, lack of ambiguity.
5. Analyzability by mechanical means - increases according to the formalness of the language.
6. Testability - provides procedures that verify if the design satisfies the requirements.
7. Traceability - provides the capability of cross referencing the design to the requirements.
8. Executability - capability of constructing a simulator from the requirements.

9. Tolerant of temporary incompleteness - so that the requirements can evolve.
10. Adaptable to changes in the nature of the needs.
11. Economy of expression.
12. Modifiable.

Teichroew and Hershey [18] advise including a glossary in the requirements specification. This provides a common basis for communication.

2.2.5. INTERFACE REQUIREMENTS

A detailed functional specification for the external interface should be prepared after requirements and before design. [19] This document provides input for the user manual and the design specification. It specifies syntax and semantics of all user inputs, functions to be performed in response to inputs, error messages, and other inputs and outputs including files and data bases. This user manual provides feedback to the user as to what the system will look like, showing the developers' interpretation of the requirements. Because the manual includes all details of functionality, it provides a basis for the estimating process. It gives exact specifications for the implementors, and can also be used for development of specifications for testing.

2.2.6. REQUIREMENTS PRESENTATION

Requirements documentation should be presented from different perspectives for different audiences. It should include several forms of representation: natural language, graphic representation, formal notation of processes performed upon data. Natural language is for the client; formal language is for the rigorous specification. The client document should be short. The advantages are brevity and understandability. The full requirements document must be more lengthy to provide precision and rigor. [20] The documentation should be readily accessible so that it can be reviewed and modified.

SECTION 3. DESIGN SPECIFICATION

Design specification describes system architecture. First, the architectural design is developed. This is a preliminary or general design of the system structure at the module level. In the second phase, detailed design, algorithms to implement each module are developed, providing a procedural description of the system. "Top level design of a software system is currently more art than science. The choice of the design approach often reflects the experience base and preference of the designer." [1]

A broad survey of software design techniques is presented by Yau and Tsai. [2] They categorize architectural design techniques as process-oriented and data-oriented.

Process-oriented techniques emphasize structure and the process of decomposition in creating architecture. Modular programming technique builds a system out of small independent modules; each module fills a single function. Functional decomposition uses information hiding as a criterion to decompose the system through stepwise refinement. Communication of system elements is through well defined interfaces. Data flow design methods use information flow to develop the system. Two data flow techniques are structured design, in which the transformation of data flows is the central focus, and SADT. Data structure design methods emphasize the structure of the problem; architecture and detailed design are developed concurrently. Jackson Structured Design and the Warnier Methodology identify and diagram the structure of the input and output data, then derive and allocate operations for the program from the merged structure of the input and output data. HIPO (Hierarchy plus Input Process Output) is a documentation tool that represents relationships between input/output data and process. It decomposes the system in a hierarchical way without involving logic details. Modular interconnection languages specify module attributes and interconnections among modules in large scale systems.

Data-oriented design techniques emphasize the system's data components and the derivation of the data design. Object-oriented design creates abstract data types and maps the problem structure into these data abstractions rather than into the control and data structures of a programming language. Conceptual

database design methodology is a guide to translate data and requirements specifications into a database conceptual schema. The software design process is viewed as a process of building a data model. One method is based on progressing from the general to the specific. General classes of data and events are defined. Then, in successive iterations, subclasses of data and specialized transactions are described until the fundamental objects in the class are reached.

Detailed design techniques include code level methods, especially structured programming, and graphical and language techniques for representing the design. In addition to flow charts, which are considered to have insufficient notation to design large-scale systems, Yau and Tsai describe Nassi-Shneiderman Diagrams (N-S Diagrams) and Hierarchical Graphs (HG's). N-S Diagrams provide special rectangular diagrams for structured programming constructs. Unlike flow charts, they do not permit arbitrary transfer of control. And they illustrate the scope of local and global data. HG's model programs and data structures using directed graphs of control flow, data flow, and program objects. The graphs provide a high-level view of program code for users and maintainers. Language representation techniques are not discussed at length in the survey. Only Program Design Language (PDL) is mentioned.

3.1. PURPOSE

The design specification fills a critical role and purpose in the development cycle. It is a medium of communication directed both to the developers and to the client who will utilize the system. Graphic models of the design, such as structure charts and data flow diagrams, facilitate communication. They make it easier to grasp the underlying structure of the system. And they provide a useful summary of the basic relationships of the system elements. They are the system blueprints.

3.2. RELATION OF DESIGN TO REQUIREMENTS

The design specification functions as a backward reference to requirements. Each functional requirement should be related to a design element that fulfills it; each module should be related to a functional requirement.

The design specifies the solution to the needs expressed in requirements.

Where requirements specify the functional and performance characteristics independent of any realization, design specifies the actual structure and behavior of the system that implements the requirements.

3.3. RELATION OF DESIGN TO IMPLEMENTATION

The design specification functions as a forward transformation to implementation. Requirements should be traceable through the design to the implementation and test plan.

The design specifies a higher level description of the product. It is less refined than the final realization of the solution, but it expresses exactly the same system as the implementation. There should be no conflict between the two, and no significant additions to the design should be present in the implementation.

3.4. ROLE OF LANGUAGE IN DESIGN

The language used to express the design affects the analyst's approach to the problem. It provides a range of expression but at the same time constrains thought patterns. [3] The design language must provide constructs to express the problem solution. It must reflect our view of the abstract world. And it must be extensible, must provide a mechanism to create higher level abstractions from the elements provided in the language.

The language affects the ability of the design to function as a medium of communication. Natural language may seem more understandable at first, but ambiguity may cause it to be less understandable when subject to scrutiny. A formal language may be processed by a computer to provide organization and cross referencing that facilitate comprehension of the design.

The language determines the extent to which the design may be validated. In order to determine whether the software meets the specifications, the design must be testable. For testability, the language must be specific, unambiguous, and quantitative wherever possible. Vagueness should be eliminated so that testing procedures may be developed during the design phase. Test design raises questions about the system design which, when resolved, increase the understanding and result in a stronger design. The benefits of the test design process can be

realized fully when the system is still in the design phase. When test design is deferred until the implementation or testing phase, it is too late to apply the increased understanding to the completed system design.

3.5. OBJECT-ORIENTED DESIGN

Object-oriented design is based on the concepts of information hiding and abstract data types as articulated by Parnas and Guttag, respectively. [4]

> "An object is a uniform representation that is an abstraction of the capabilities of a computer to store information. An object has the capacity to store information; we say that an object has private memory. An object also has the capacity to manipulate its stored information to carry out some activity. These are called the operations of an object. The set of operations is referred to as the object's interface. A crucial property of an object is that its private memory can only be manipulated by the operations in the object's interface." [5]

Object-oriented design creates abstract data types, or objects, from entities in the real world of the problem. It models the system around the objects and the operations that characterize them, concentrating on the design rather than details of the objects. Because of its strong data typing, and its encapsulation features, packages and tasks, Ada is well suited to object-oriented design.

According to Boyd [6], the underlying principles of object-oriented design are:

1. Information hiding.
2. Abstract data types.
3. Characterization of system components as objects.
4. Mapping the problem domain onto user-defined constructs rather than onto predefined constructs of the implementation language.

Boyd identifies the central premise of object-oriented design: "problem definition is the first and hardest task a designer confronts." The object-oriented design method enables the designer to clear away what is unnecessary, to allow critical features to be generalized into a model of the real-world situation.

3.5.1. STEPS IN OBJECT-ORIENTED DESIGN

There are well-defined steps in the object-oriented design methodology. [7]

A. Define the problem. A context diagram is a useful aid.

B. Develop an informal strategy for the abstract world. This may be a general sequence of steps that satisfy the requirements.

C. Formalize the strategy:

   1. Identify objects and their attributes.
   2. Identify operations on the objects.
   3. Establish the visibility of each object in relation to other objects.
   4. Establish the interface of each object.
   5. Implement the operations.

The design process is recursive because implementing operations reveals hidden objects at lower levels of abstraction. [8]

Chen and Steimle [9], like Booch [10], identify objects and operations by extracting noun and verb phrases from a statement of the informal strategy. The nouns become the identifiers for the objects, and the verb phrases become the operations.

3.5.1.1. IDENTIFY THE OBJECTS

Objects are classified into 3 types: actors, servers, and agents. [11] An actor undergoes no operations; it only operates on other objects. A server only undergoes operations and cannot operate on other objects. An agent is an object that serves to perform some operation on behalf of another object and in turn can operate upon other objects.

Seidewitz and Stark [12] discuss the process of abstraction analysis. An abstraction's strength is according to the amount of detail that it suppresses. "Good" objects closely model entities in the problem domain; they are characterized by abstraction and information hiding. Other abstractions have little reason for existing. Object abstraction types, from best to worst, are:

1. Entity abstraction. The object represents a useful model of an entity in the problem.
2. Action abstraction. The object provides a generalized set of operations which all perform the same kind of function.
3. Virtual machine abstraction. The object groups together operations which are all used by some higher level of control, or which all use some set of operations on a lower level.

4. Coincidental abstraction. The object packages a set of operations which have no relation to each other.

Abstraction analysis is a method of transforming a structured specification into an object-oriented design. Steps in the design process are as follows.

1. Find a central entity from the top level data flow diagram by identifying a set of processes and data stores that are most abstract.
2. Find entities that directly support the central entity.
3. Follow the data flows and identify entities until all the processes and data stores are identified with an entity.
4. Construct an entity graph showing processes and data stores in entity squares and indicating the initial flow of control.
5. Construct an object diagram showing the flow of control and the virtual machine levels.
6. Identify the operations provided by and used by each object by examining the data flow.
7. Document the object description.
8. Using the subset data flow diagram of processes and data stores, produce child object diagrams and identify entities based on how they support the parent object's operations.
9. Transform the object diagram into Ada. Package specifications are derived from the list of operations provided by an object.
10. Package specifications for the top level object diagram are placed in the declarative part of the top level Ada procedure.
11. Package specifications of lower level objects are nested in the package body of the parent object.
12. Follow this procedure down to the level of implementing subprograms as subunits.

Elements of the object diagram transform directly to Ada constructs. An object becomes a package or task. A procedure is a subprogram. A state is a package or a task variable. An arrow in the diagram becomes a procedure or function or a task entry call.

3.5.1.2. IDENTIFY THE OPERATIONS

For each object identify the operations that affect it and the operations

that it initiates. "The operations suffered by each object from within the system . . . roughly parallel the state change caused by a data flow into an object. . . The operations required of each object roughly parallel the action of a data flow from an object." [13] The system should have a balance of operations suffered by and required of all objects. For each operation permitted by an object, there is another object that requires it. Localizing the required operation with the object of which it is required decouples the two objects. The object with its associated operations is independent and reusable. [14]

3.5.1.3. ESTABLISH THE VISIBILITY

To establish visibility is to indicate dependencies among the objects. Dependencies follow the direction of the operations required of an object. "B is visible to A" means that A sees B; A depends upon the resources of B; A requires an operation of B.

To graph the relationship, we would draw an arrow from inside A to the outside edge, the interface, of B.

3.5.1.4. ESTABLISH THE INTERFACES

The interface forms the boundary between the outside view and the inside view of an object. The interface "captures the static semantics of each object . . . and serves as a contract between the clients of an object and the object itself." [15]

3.5.1.5. ADA FEATURES FOR OBJECT-ORIENTED DESIGN

Ada has elements of all three generations of programming languages: (1) tools for mathematical expression; (2) tools for algorithmic control through structured language constructs; (3) tools for the expression of data structures. In addition, it has tools to enforce abstraction. [16]

Ada enforces abstraction through private and limited private data types. It provides information hiding in the package body. An object can thus be made available while the implementation is inaccessible. By restricting the visibility among objects, packages limit the number of objects we must deal with to

understand any part of the system, reducing complexity. By localizing design decisions, packages limit the scope of change. [17]

Object-oriented design encourages exploitation of Ada features such as packages and tasks. "Packages represent a logical collection of computational resources that can be used to encapsulate: (1) a named collection of declarations; (2) a named collection of subprograms; (3) an abstract data type; (4) an abstract state machine." [18]

3.5.2. OBJECT-ORIENTED DESIGN AND TRADITIONAL DESIGN

Traditional design methods are either process-oriented, in which data tends to be global, or data-oriented, in which processes are treated globally. Object-oriented design differs from traditional design methods. It is neither data nor process driven and treats neither of these globally. It provides a balanced treatment of data and processes. [19]

Traditional, top down design techniques are imperative in nature. They decompose a solution functionally, into sets of procedural modules that represent abstract actions. Procedures can not model an entity with memory. Normally, global data is used for memory, with data visible to any level of the system. Changes in the problem cause data changes that ripple through the entire system structure.

Object-oriented design views modules as collections of computational resources representing abstract data types and abstract operations. Each module represents an object, not a step in a process. Data is restricted, encapsulated in modules, rather than global. The system is resilient to change because the effects of change are localized.

Booch points out differences in the structure of systems developed with object-oriented design. (1) Components tend to form a directed acyclic graph rather than to be strictly hierarchical and deeply nested as in systems developed traditionally. In object-oriented systems many threads of control may be active simultaneously, rather than a single thread following the hierarchical lines of decomposition. (2) The subprogram call profile of object-oriented systems typically exhibits deeply nested calls as objects invoke operations upon other objects. [20]

Programming with abstract data types is not the same as object-oriented

design, according to Booch. [21] (1) Development with abstract data types tends to deal with passive objects (agents and servers). It is inadequate for problems with natural concurrency. Object-oriented design also concerns itself with actors, modules that need no stimulus from other objects. It is suited for problems involving concurrency. (2) Development with abstract data types deals with the operations suffered by an object. Object-oriented design also concerns itself with the operations that an object requires of other modules.

3.5.3. WEAKNESSES OF OBJECT-ORIENTED DESIGN

Booch cautions us that object-oriented design is a partial life-cycle method; it focuses on design and implementation. To be fully effective, it should be coupled with a requirements method to provide a strong foundation for the design. [22]

Other weaknesses are discussed by Boyd. [23]

(1) Transitions in the method may lead to a series of denotations with significantly different informational content, resulting in possible loss of information. For example, data flow direction is not explicitly carried through.

(2) There is a single representation for two independent issues: data transfer and call decision. It does not support the independence of these issues in the denotation. The component initiating a data transfer is not necessarily the component from which the data is transferred.

(3) Concurrency is not well denoted. There should be representations for task priorities, task types and rendezvous semantics. "Very little progress has been made in logical proofs for asynchronous abstract data types; this may account for the lack of robust exploitation of tasks in object-oriented design."

(4) The method is biased toward representing a system under design rather than in operation. "Object-oriented design's emphasis upon the definition of objects in a static dependency network restricts its suitability for the dynamics of Ada systems in operation."

(5) There is no means to specify the system behavior at an abstract level. In object-oriented design, system components are linked by resource dependencies rather than behavior.

(6) Object-oriented design does not represent data flow, since only resource dependencies are represented. Data flow is only represented after the

Ada program design language is developed "and that description will be textual when present at all."

3.5.4. STRENGTHS OF OBJECT-ORIENTED DESIGN

Yau and Tsai mention the advantages of productivity, maintainability, data integrity and program security; these result from object-oriented design's characteristic data abstraction, program abstraction and protection domains. Productivity results because the abstraction mechanism allow the construction of reusable components. Maintainability is achieved by information hiding, which localizes modifications within a component. Data integrity and program security are achieved by the protection domains which define access rights and operations available to a component's user. [24]

Understandability results because the structure of the design corresponds to the world that it models. Understandability is referred to by Boyd [25] when he points out that object-oriented design permits collections of objects to be grouped into subsystems to denote logical relatedness. The method therefore provides a layered approach that well serves the design of large systems. The result is a relatively flat system rather than a deeply nested hierarchy.

3.6. COMPARISON OF OBJECT-ORIENTED AND FSM DESIGNS

Chen and Steimle compared an object-oriented design and a finite state machine design for the same concurrent system. [26] They designed a subnet layer service between communications media and an encryption device, using Grady Booch's methodology as outlined in Software Engineering With Ada. The two designs were expressed in Ada and evaluated according to performance, portability, and reusability.

They found that the finite state machine design would have high performance efficiency. There were no task rendezvous in the FSM design, where there were three in the object-oriented design, resulting in much lower efficiency.

The object oriented design would have high portability and reusability. For the finite state machine design portability and reusability are restricted. To avoid rendezvous, the FSM design used a global wait for completion of service, for which no facility is provided in the Ada language. For each unique system, a

system interface package would have to be written in assembler, C, or some other target system language. The object-oriented design would be reusable without recompilation. For the FSM design to be reused, "modification, recompilation, and careful use of generics would be required."

The authors conclude that the immediate need for efficiency should be balanced against the long range need for portability and reusability when the design method is chosen.

SECTION 4. OBJECTIVES OF THE DESIGN SPECIFICATION

Design captures the state of the system in development midway between requirements and implementation. The objectives for the design specification reflect the utility of the design as a transition and the utility of the system expression at the transitional level.

4.1. FORWARD TRANSFORMATION TO IMPLEMENTATION

The design must be useful as a forward transformation to implementation. The design specification is the blueprint for the system. It specifies the structure of the data and the architecture of the processing system at the modular level. In addition, it describes algorithms to various levels of abstraction/detail as appropriate. The design must express the mechanisms to produce all behavior specified in the requirements. [1] It replaces the requirements specification to become the rules for the next phase of system development.

4.2. BACKWARD REFERENCE TO REQUIREMENTS

The design must be a valid backward reference to the system's requirements. Because it replaces the requirements as the rules for later stages of development it must completely express the requirements at the new level of design. It must include required system objectives that are not obviously related to the constitution of the designed product. [2]

4.3. SOFTWARE DOCUMENTATION

The design specification functions as the documentation component of the software. It provides documentation for implementing, verifying, utilizing, and maintaining the product. It is more informative than the functional specifications because it is more detailed; it is more understandable than the implementation because it is less detailed. In addition, it relates the system's behavior to its architecture.

4.4. DESIGN PROPERTIES

Before implementation, the design must be evaluated to determine if it meets design objectives. Yau and Tsai name properties to be validated in software design. [3]

1. Completeness. The design fully develops each element of the requirements specification.
2. Consistency. No conflict exists between portions of the design.
3. Correctness. The input and output relation can be proved true or false.
4. Traceability. Terms in the design have antecedents in the requirements specification. Nothing has been added that is not required.
5. Feasibility. The design can be implemented and maintained so that the life-cycle benefit exceeds the cost.
6. Equivalence. Two equivalent designs have the same behavior.
7. Termination. The design is sufficiently detailed for implementation.

Booch discusses design principles according to Ross, Goodenough and Irvine. [4] In order to achieve the software engineering goals of modifiability, efficiency, reliability, and understandability, a good design should manifest:

1. Abstraction - extract essential details at an appropriate level.
2. Information hiding - make inaccessible all details that don't affect other parts of the system.
3. Modularity - purposeful structuring.
4. Localization - group logically related resources in one physical module.
5. Uniformity - consistent notation, free of unnecessary differences.
6. Completeness - all important elements are present.
7. Confirmability - can be readily tested.

Booch emphasizes that the current standard of functional top-down design supported by structured programming does not adequately meet these goals. It does not support information hiding and enforce abstraction, and does not control the complexity of data structure design. [5]

SECTION 5. OBJECTIVES OF A SPECIFICATION LANGUAGE.

A program design language (PDL) is the means of recording the design. Its purpose it to provide a medium for communicating and verifying the design. It is the basis for design reviews and the repository for design history. The language chosen for the design should support the design discipline or methodology. [1]

5.1. FEATURES OF A SPECIFICATION LANGUAGE

A specification language must have features to express all the elements of the design. [2]

1. Data definitions - sufficiently powerful to express the system state in all areas of the problem within its intended scope. It should include declarations of scalars and data groupings such as records and arrays, plus user-defined data types.
2. Structural objects - subsystems, modules, and files.
3. Behavioral abstractions - to define system laws, details, and algorithms.
4. Sequentiality and concurrency - ability to design for several inputs producing several outputs in an unspecified sequence.
5. Interactions between objects - interfaces, connectivity, and behavior of design elements.
6. Error conditions and recovery.
7. Modularity - design components of system separately as procedures and functions.
8. Generality - reusable for different systems.
9. Interfaces with the user and environment.
10. Performance standards.
11. Constraints.

The last two features pose a major problem for formal specification languages - the expression of non-functional requirements. Roman states that the major difficulties in expanding the scope of specification techniques are (1) establishment of a formal foundation for the non-functional requirements and (2) broad integration of the functional and non-functional requirements.

> "Any attempt to separate requirements and design specifications is counter-productive when one deals with the design stages. The key to across-life-cycle integration of design activities rests with the ability to relate design and requirements specifications. . . Consequently design languages must have the ability to specify the requirements for the types of subcomponents they identify and they must overcome the current emphasis on functionality alone by incorporating formally an increasing number of non-functional requirements." [3]

Roman reminds the reader that the severity of the constraints affects the complexity of the design. Much design effort is expended in checking whether the constraints are met.

Roman, too, stresses traceability. Loss of requirements traceability causes maintenance problems because it is then impossible to tell if a function is required or if it is a design constraint that is no longer needed. Without a statement of the requirements' purpose, the designer may solve the wrong problem.

Further requirements are added by Sammet, Waugh, and Reiter. [4]. The language should impose structure while permitting free-form expression of specific application ideas. It should provide formal commentary with specified format and scope.

Nejmeh and Dunsmore [5] recommend that (1) the program design language should support software design for a number of implementation languages. The modules, data, and control flow constructs should be programming language dependent in order to verify the interfaces, to make accurate metric estimates, and to promote structured coding. But the language should not support low level detailed constructs. (2) The language should provide libraries of formally defined design constructs and a mechanism to present online specification of functionality and required environmental conditions for the construct to function properly. It should be possible to compose a new system from modules constructed in different programming languages. The environment should provide a means of transforming high level abstract design constructs into the implementation language code. (3) The language environment should provide tools for simulation of the design, for producing the design graph showing the module interconnections, for listing TBD (to be determined - unfinished) constructs in a deferred implementation report. There should be a mechanism for showing the mapping from requirements to design specifications. Another useful tool would be a list function with design expansion to the low level turned on or off - on for walk-throughs, off for

managers and non-programmers.

The relation of the design language to the implementation language is a matter for debate. Nejmeh and Dunsmore [6] disagree with the above authors. They recommend that the design language should be independent of the programming language in order to avoid the tendency to be too detailed in the design. However, the language should promote structured coding in implementation and should have good code-to ability. The primary characteristic of a program design language is that it possesses a fixed syntax: a fixed syntax of keywords providing structured constructs (data declarations and modularity characteristics) and a fixed syntax of design constructs used to convey design ideas.
It should support the use of tools, be automatically translatable to code, and promote the ability to compute metrics.

The following features are noted by Anderson. [7] The language should be human engineered so that it is "usable as an aid . . . and not an extra nuisance." It should support a variety of development methodologies, including early prototyping. It should permit "selective focus" so that the programmer can view the system from the topmost level and can abstract patterns from the complexity of the software. Anderson points out the advantage of a standard design language for portability of programs and programmers.

5.2. SUPPORT FOR MODERN PROGRAMMING PRACTICES

A specification language should support modern programming practices which control complexity. [8]

1. Abstraction - to extract essential concepts while suppressing unnecessary details.
2. Decomposition - division into smaller, more manageable pieces while maintaining a fixed level of detail.
3. Information hiding - isolation of unnecessary details.
4. Stepwise refinement - progressive addition of detail.
5. Modularity - development from standardized units.

The design language should facilitate following these practices so that they will be used naturally and routinely in system designs.

5.3. IEEE RECOMMENDED PRACTICE

The IEEE Recommended Practice for Ada as a Program Design Language (Standard 990-1987) includes additional objectives for design languages. [9] According to this document, the language should be able to convey product information and management information. Product information includes performance, security, fault tolerance, traceability and other standards. Management information includes: (1) organizational information - e.g., tasks assigned to team members; (2) planning information - milestones, resources, dependencies; (3) status information - milestone completion; (4) configuration management information - configuration identification and change control restrictions.

IEEE discusses two characteristics of an Ada program design language, (1) conformance to the language standard and (2) language extensions. The program design language is in conformance with the Ada language if it is compilable on a validated Ada compiler without error. It could be a subset of Ada; this would prevent untranslatable constructs, inefficiencies and excessive detail.

IEEE would permit extensions of Ada through structured and unstructured comments, provided that the comments include only information that can not be reasonably expressed by Ada constructs. Unstructured comments would be used for natural language explanations of the design. They would also be used for information needed in the design process where formal structures are not required, such as in "human-to-human communication." Structured comments would be used to provide design information in "additional design-oriented semantics." Such structured comments should be consistent with the general syntactical structure of the Ada language. They should be identified by a sentinel character immediately following the double dash that indicates a comment in Ada.

5.4 MEDIUM FOR COMMUNICATION

The language should be an unambiguous, effective medium for communicating the design. It must be suitable for comprehension by designers, reviewers, programmers, maintainers, managers, and quality assurance personnel. And it must convey complete design information to suit the needs of each of these groups. The design document should be the single source for all required documentation.

Language qualities that relate directly to communication are discussed by

Rodney Bond. [10] He favors an English-like pseudo-code that supports productivity, top-down development, standardization, and analysis. It should provide for flexible pseudo-code definitions and should restrict the definition to meaningful information at the design level. It should support progressive refinement by allowing for incomplete descriptions; it must include a TBD construct. Finally, the language must be suitable for automated tools that enhance communication by providing organization: pretty-printers, cross-reference lists, calling trees.

5.5 DIFFERENCES BETWEEN DESIGN LANGUAGE AND HIGH LEVEL LANGUAGE

A design language differs from a high level programming language. Sammet, Waugh and Reiter draw the following distinctions. [11] (1) A high level language must be executable; a design language is generally not executable. (For a design to be executed, it must be simulated.) (2) A high level language is fully analyzable by computer; a design language may contain constructs that are only partially analyzable by computer. (3) A high level language must be rigorously defined; a design language may be loosely defined. (4) A high level language usually supports only low levels of constructs and abstractions; a design language supports varying levels of abstraction. (5) A high level language product is usually subject to configuration management; a design language product must be easily changed. (6) A high level language is intended for machine communication; a design language is intended for human communication.

The difference between Ada and an Ada program design language lies in language extensions to support program design. Weissnesee stresses the need for language extensions when he says, "The practical requirements and scope of an Ada or Ada based Program Design Language greatly exceed that of the language itself." [12] He points out the productivity advantage of designing in the superset language (Ada with extensions) vs. designing in pure Ada. [13] Compilable code is a by-product, not the object of the design. Weissnsee's experience is that 5-10% compilable code is produced as part of preliminary program design; 15-50% compilable code is produced in detailed program design. "Any overall software coding percentages greater than 50% leave a question of 'Are you designing or are you programming?'"

This becomes an important question when pure Ada is used for design. In that case, the high level language and the program design language are the same. Care must be exercised to avoid coding and stick to design. This may be difficult when the language requires expression of necessary details using low level constructs.

SECTION 6. FORMAL LANGUAGE AND NATURAL LANGUAGE FOR SPECIFICATION

The problem in choosing a specification language is to achieve a balance between natural and formal language that is appropriate for the situation. Natural language alone is imprecise and not machine processable. Formal language alone is difficult to create and maintain. In order to be fully executable, it must be refined down to the level of code, i.e. must specify the implementation. [1]

Program design languages lie somewhere between natural language and high level programming languages such as Ada. A program design language is "a tool which uses the vocabulary of a natural language and much of the syntax of a structured language" such as Pascal. It is "structured English" which allows the specification of algorithms and data structures. [2] Like any compromise, it provides some of the advantages and disadvantages of each alternative but the full benefits of neither one.

6.1. ADVANTAGES OF FORMAL LANGUAGE

Formal language provides important advantages. Its use is advocated to fulfill a variety of purposes.

Formal notation raises design quality by eliminating ambiguity and inconsistency. It focuses attention on the problem in a structured manner, eliminates ambiguity from the communication of the problem, and provides a framework for expressing solutions. [3] When design constructs are more easily understood, the mapping from requirements to design can be more closely monitored. Criticisms can be obtained early in the software life cycle. Increased system reliability results because the ability to draft test plans is enhanced by understandability. [4]

Formal language decreases coding, testing, and maintenance effort by improving design quality. Only a part of the decreased effort from these phases is shifted to the requirements and design processes. Errors are detected and corrected in the design stage because of increased precision in

the design, and because of the use of tools for cross-checking the design. The compiler enforces syntax and does type checking. In addition, the design is more easily implemented because the formal language of the design is closer to code than is natural language.

Reusability of design constructs is promoted by using formal language. In order to be reusable, there must be a way to precisely characterize the component so that it can be determined if the component meets the need. "In the presence of ambiguously specified components, most clients will give up searching and will build their own, thus destroying all hope of reuse." [5] Formal language for specification not only provides a precise characterization but also facilitates component retrieval because it is processable by tools.

Using formal notation enables use of tools for verification and documentation of the design. When programs are specified informally, it is impossible for tools to extract the ideas underlying the program algorithms and verify the program's correctness. When specification is expressed in a formal language, mechanical tools can carry out each development step, guaranteeing correctness. [6] Documentation tools can extract and organize a variety of information if it is expressed formally using key words or symbols to characterize the information content of parts of the specification.

A principal advantage of formal language is that it promotes automatic transformation of design to implemented code. As a result, errors in specification are found early in the requirements phase, not after design and implementation. No errors are introduced in the transition between development phases. Software costs are controlled. Because requirements are modifiable and code is automatically generated, it again becomes economical to customize software rather than settle for purchased software that may not fully meet the client's requirements. [7]

Formal language enables early and accurate estimation of software metrics. McCabe's complexity metric V(G), calculated from a formal design, can be a basis for estimates of testing effort. If less than 10% of the instructions are conditional transfers of control, productivity will be high because of less required testing effort. [8] Another metric, Design Completeness (DC), is the ratio of a design metric to the actual value of the implementation metric. As the ratio approaches one, the design completely expresses the implementation. DC is an indicator of design quality that may be calculated when formal

language is used for design.

Using a formal language can "enforce a rigorous methodology for system development." The need for rigor in requirements specification was a force in the development of the formal language IORL. [9] Jackson [10] advocates using a rigorous development methodology such as VDM to cope with the problem of Ada complexity. A rigorous method would lead to a disciplined use of Ada features. The result would be reliable systems.

Formal language permits traceability of requirements throughout a project. Performance of maintenance upon formalized requirements or design specifications, with computer generation of code, guarantees that the implementation remains true to the specifications. As a result, system documentation is always up to date.

6.2. ADVANTAGES OF NATURAL LANGUAGE

Informal specifications can be more concise, readable, and understandable. "They are concise because only part of the specification is explicit; the rest is implicit and must be extracted from context. Attention is focused on the explicit information and, therefore, away from the implicit information, which increases both the readability and the understandability of the specification." [11]

Informal specifications use the normal mode of communication. No training is needed to produce the specifications or to understand them. They communicate immediately to mangers, developers, and especially clients. Smoliar and Barstow are concerned that formal language isolates the end-user from the development process. [12] They stress that the environment interfacing with the client should use natural language to permit the user to focus upon the problem domain rather than the language.

Informal specifications are easily maintainable because they are less complex. Information is not spread throughout the system. "The creation of a formal specification involves spreading implicitly specified information throughout the specification and increasing the complexity by structuring the specification into parts and establishing the necessary interfaces between them." [13]

6.3. AUTOMATIC TRANSFORMATION FROM NATURAL TO FORMAL LANGUAGE

A tool to transform informal specifications to formal language would be a compromise that would provide the advantages of both formal and informal language.

An artficial intelligence based tool with expert knowledge of the problem domain would permit the user to interface with the system without mastering the syntax and semantics of program constructs. Smoliar and Barstow describe their automatic programming project within the domain of quantitative log interpretation for petroleum science activity. [14]

Balzer, Goldman, and Wile discuss their work in resolving ambiguity in order to transform informal language to formal. They have developed a prototype tool called SAFE, that utilizes context to complete informal and partial specifications, using a relational data base approach. They feel that the partial descriptions focus attention on the relevant issues and condense the specification. The tool provides for feedback and interaction with the user to eliminate the problem of possible misinterpretation of the informal specification. Once the functionality is accepted by the user, the system restates it as a formal operational specification and determines the final program structure. [15]

Comer describes a processor that accepts descriptions of data structures in natural language. This has been successful because only a limited subset of English is required to describe data types. The processor automatically generates data type specifications, including operations on the data type, input to the operations and output, as well as exception conditions. It also produces an axiomatic description of the data access operations. [16]

6.4. FORMAL FOUNDATIONS FOR SPECIFICATION LANGUAGES

A number of formal foundations provide alternatives for specification techniques: finite state machines, data flow, stimulus-response paths, communicating concurrent processes, functional composition, or data-oriented models. Roman discusses formal foundations and provides an example language for each one. Below are brief descriptions of these foundations, taken from Roman. [17]

A technique that uses finite-state machines "treats system processing as a mapping that takes the current system state and an incoming stimulus and produces a new system state and a response."

A data flow model "consists of processing activities and data arcs showing the flow of data between the activities. Processing is triggered by the presence of data in the input queues associated with each activity."

"Techniques using stimulus-response paths decompose the requirements with respect to the processing that must be carried out subsequent to the receipt of each stimulus."

In all techniques using communicating concurrent processes, "a process is represented by a set of states and by a state transition mapping." The techniques differ mostly in the way in which the processes communicate and exchange data. They may use application of functions, queues, or sets of primitives designed for communication.

Using the functional composition approach, the user graphically "define[s] the system's functionality as a composition of mathematical functions."

"Data-oriented techniques concentrate on the specification of the system state represented by the data that needs to be maintained." In Roman's example, "the system functionality is defined in terms of built-in data manipulation primitives; other techniques provide the means to define system activities in a manner similar to that of defining data objects."

6.5. SEMANTIC MODELS FOR FORMAL LANGUAGES

Semantic models for formal languages may be conceptual, denotational, axiomatic, operational. Roman recommends that persons involved in design activity understand the principles behind semantic models to help them interpret requirements written in languages based upon these models.

In conceptual modeling, the meaning of a program is expressed in terms of entities in the real world, rather than in terms of data records. Relationships are expressed as properties of objects. Objects are arranged in abstraction hierarchies. As a result, focus is on the problem rather than on implementation issues. [18]

Using the denotational model, the meaning of a program is stated as a mathematical function. The function at the highest level can be represented in

terms of P: input + program process = output.

Using the axiomatic model, the meaning of a program is stated by providing the axioms and inference rules needed to prove the program is correct. Abstract objects are specified by sets of axioms relating the operations permissible for each object. Procedures are specified by assertions about the input and output.

Using the operational model, the meaning of a program is given by the result of executing it on an abstract machine. Abstract objects are represented "by showing how each operation uses and modifies some abstract representation of the object. " Procedures are represented by "simple and clear algorithms that perform the same function as the intended program but ignore any performance issues. (These algorithms are not intended for use by the actual program.)" [19]

SECTION 7. FORMAL SPECIFICATION LANGUAGES

This section provides brief descriptions of some formal languages used for specification. Some are called requirements languages and others are called design languages; all express system design.

> "Traditionally, requirements writers have seen themselves as first-level system designers; it was their responsibility to digest a customer's needs, and to produce a preliminary system design. . . A large number of so-called requirements tools primarily support design (i.e., a decomposition process) but also support a limited requirements specification facility at each level of design. These tools include . . . PSL/PSA . . . SADT . . . IORL. . . There are only two requirements tools currently in use which are purely requirements tools. These are REVS [SREM] . . . and RPS [RLP]." [1]

7.1. USE.IT

USE.IT, from Higher Order Software (HOS) is a functional composition language that generates code for the implementation.

> "Here, useful functions are selected, relationships between these functions are determined (and resolved if they are inconsistent), and redundant functions are eliminated. Once this process has been performed it is easier to get an idea of what functions are missing. Although one could interpret such an approach as a relational one, the relations between functions can ultimately be understood in terms of functions." [2]

HOS claims that designs specified through USE.IT are provably correct because all control structures and defined operations are built out of primitives to which mathematical proofs of correctness apply. The language uses 3 primitive structures: JOIN, INCLUDE, and OR; an additional 4 co-control structures are derived from the three primitives: COJOIN, COINCLUDE, COOR and CONCUR. These 7 control structures
are founded on 6 basic axioms that provide rules for the decomposition of systems: [3]

1. Invocation - A parent can invoke only its immediate offspring.
2. Responsibility - A parent is responsible for producing output data types with correct values.

3. Output access rights - A parent can assign its offspring the right to alter the parent's output variables.
4. Input access rights - A parent grants its offspring the right to access the parent's input variables.
5. Rejection - An offspring rejects input that is not the input of its parent.
6. Ordering - The parent controls the order of invocation of its offspring.

USE.IT is a family of tools that automate a functional life cycle model consisting of 6 major functions: [4]

1. Manage - Integrate the relationships between the next 5 processes (with the tool USE.IT).
2. Define - Describe the required system (with the tool AXES).
3. Analyze - Test the system by relating it to a set of instances - what if's, (with the tool ANALYZER).
4. Resource allocate - Implement the system by relating it to a machine architecture (with the tool RAT).
5. Execute - relate the system to instantiations (using the target machine).
6. Document - relate the system to a communication vehicle.

USE.IT manages 3 other tools for accomplishing functions 2 - 4. Each tool and USE.IT fulfill function 6 by providing self-documentation. [5]

AXES is an interactive tool that assists the user in defining system data types, functions and structures in graphics or statement form. AXES is a language, but not a programming language. The language is non-procedural; the user can use her/his own syntax. It is a language for defining mechanisms for defining systems. The definitions adhere to AXES semantics, but the syntax is up to the user. There can be a library of AXES definitions so that new definitions can be derived from existing ones. The AXES specifications can be translated to other representations such as data flow diagrams or structured design diagrams. A set of AXES statements is implementation independent; it allows for many options of implementation.

ANALYZER interacts with the user if there are errors in the defined system. This tool detects missing functions or missing data. It guarantees that the hierarchical definition stops at primitive operations on algebraically

defined data types. It enforces correct interfaces and correct data flows. It integrates system modules by checking across independently developed modules and checking definitions of library modules. ANALYZER ensures that the requirements are unambiguous (consistent). After successful analysis, the requirements are free of interface errors and are the same ones that the user defined.

RAT is the Resource Allocation Tool; it transforms the AXES specification to source code after it has been successfully analyzed by ANALYZER. RAT generates simulations from control maps involving unimplemented primitive operations. From control maps involving implemented operations, it generates efficient implementations. It permits inclusion of existing higher order language packages into the user library as external operations. RAT can generate code for the same definition in different languages: FORTRAN, Pascal, COBOL, Ada. The code generated by the RAT is ready for compilation and then execution.

Documentation is produced by USE.IT, AXES, and the RAT. USE.IT contains a plotter which produces documented plotted output of the system graphs. AXES produces a documented hierarchy of the requirements. RAT produces documented code.

7.2. PDL

PDL (Program Design Language) was an early high level design specification language (1974). It is a Caine, Farber & Gordon product.

PDL resembles structured English rather than a formal language. Its primary usefulness is as a medium of communication. "The basic readability of a PDL design means that clients, management and team members can both understand the proposed solution and guage its degree of completeness." [6] "It supports a rich set of structured constructs . . . However, the lack of formally defined design constructs implies that it is quite possible for ambiguous design descriptions to exist." [7] The lack of formality also limits tool support and prevents producing accurate software metric estimates.

PDL/81 is a software tool consisting of a processor and a data base. It "integrates the capabilities commonly associated with a program design language processor and those of a text processing system" [8] to assist in designing and

documenting a system to be developed. It processes designs and also documents such as reports and manuals.

The PDL processor executes in batch mode. It formats the design text for readability, checks for duplicate design elements, and produces data and control segment reports. It prints the design with a cover page, table of contents, reference tree and cross reference lists.

The data base is used to tailor the processor to the requirements of the particular document being produced. The project manager controls the document format and contents by composing abstract constructs from primitive formatting operations using a definition language. The designer uses these constructs to produce program design documentation without needing to understand how the constructs were created. An interactive facility permits design creation and modification online.

The designer can explicitly define data items or can use external data segments defined outside the PDL document. The processor reports data in a data index that lists references to the data items.

Procedural flow is defined in flow segments. Each flow segment represents a procedure in the program. The procedure may be entered in free form. When it prints the design, the processor underlines keywords, indents to the correct structure nesting level, and provides continuation from line to line.

Segments of text can be placed in the design to provide commentary.

7.3. SLAN-4

IBM's SLAN-4 is " a language spanning the complete range from an almost natural language to an almost compilable language." [9]

> "One of the design principles for SLAN-4 was that it should be possible everywhere to omit the formal definition and to give an informal definition as a comment. These informal parts may later be replaced by formal counterparts. However. . . each of the services a SLAN-4 processor offers (i.e. syntactical and semantical checks) depends on formal definitions." [10]

SLAN-4 features constructs for 4 approaches to software specification:

1. Abstract data types, called classes, and the operations performed on them.
2. Algebraic specifications.

3. Axiomatic specifications through pre- and post-conditions for modules.
4. Design by pseudocode.

Design with SLAN-4 is a two-step process. "The first step corresponds to an architectural and high-level design using the algebraic and axiomatic specification methods; the second step consists of a low-level design using the pseudocode part of the language." [11] Algebraic specifications are used to describe relations between modules. Axiomatic specifications are more detailed. They describe the behavior of each module through the effects of an operation with regard to its input and output state.

SLAN-4 permits the specification of sequencing constraints. It can describe concurrent processes, with synchronization control, and intermediate states of a module. A semaphore data type is provided for synchronization control.

Besides basic data types, SLAN-4 includes constructors: arrays, records, sets, lists. These may be combined in any order and to any depth. The language does not include files. The user is advised to "use the basic I/O routines offered by the implementation language or to specify what can be assumed about the I/O routines." [12]

7.4. SREM

SREM is TRW's Software Requirements Engineering Methodology. The methodology includes a language for defining requirements and a set of tools to process the language and manipulate a requirements data base.

Requirements Statement Language (RSL) "provides a checklist of characteristics for requirements specification, and it puts the requirements into a machine-readable form that can be translated into a database for automated consistency and completeness analyses." [13]

The Requirements Engineering Validation System (REVS) is the set of tools which analyzes and manipulates the requirements data base, the Abstract System Semantic Model (ASSM). The tools have six kinds of functions. [14]

1. Extension of RSL to tailor it to a specific project.
2. Translation of RSL into an automated, relational database.
3. Analysis of the database contents for consistency and completeness.

4. Extraction of information from the database.
5. Generation of simulators of the required processing from the requirements specification.
6. Generation and display of graphical descriptions of the requirements.

The SREM "approach is to define functional requirements in terms of paths of processing, then to attach performance requirements to the paths." [15]
The methodology for specifying requirements includes 7 phases. [16]

1. Define the functions identify all input, output and processes.
2. Establish the baseline - cleanup the database and generate plots of the processing paths.
3. Define the data.
4. Establish traceability.
5. Simulate the functionality of subsystems.
6. Identify the performance requirements.
7. Demonstrate the feasibility by a rapid prototype of critical algorithms.

> "The basic concept underlying SREM is that design-free functional software requirements should specify the required processing in terms of all possible responses (and the conditions for each type of response) to each input. . . Thus, functional requirements identify the appropriate stimulus/response relationships, and autonomously generated outputs. These required actions of the software are expressible in terms of Requirements Networks (R-Nets) of processing steps. Each processing step is defined in terms of input data, output data, and the associated transformation." [17]

RSL is based on a highly structured finite state machine model. [18] It uses R-nets and subnet structures to represent sequences of processing. The state is structured into sets of information about objects.

The language can express other concepts as well; it identifies source documents, requirements in the source documents, and decisions made in deriving the requirements. [19]

REVS contains tools that extend RSL for special requirements. SYSREM extends the SREM state machine model to represent decomposition and concurrency. In SYSREM, functions and performance requirements and functions are decomposed simultaneously. [20]

A second SREM extension is DCDS, Distributed Computing Design System. [21] This is a set of five languages and tools for specifying requirements involving systems distributed over multiple processors. The languages are:

1. SSL, System Specification Language - Allocates system requirements to a processor.
2. RSL, Requirements Statement Language - Defines software requirements.
3. DDL, Distributed Design Language Designs distributed processes.
4. MDL, Module Design Language - Software Design.
5. TSL, Test Specification Language - Specifies test plans and procedures.

According to Scheffer and Stone, the strength of SREM is its methodology rather than its automated tools. "The structure of the methodology guides the analyst to an understanding of the requirements through an iterative learning process that exposes the requirements flaws." [22] It provides a disciplined technique that improves the integrity of the system description by revealing omissions, inconsistencies and ambiguities.

Other benefits of SREM are discussed by Alford [23] SREM has an objective stopping point for the requirements definition. It provides verification of data flow consistency (no data may be used before it is given a value). It specifies performance in testable terms. It provides traceability of system-level requirements to the software processing requirements.

Alford points out that SREM forces expression of requirements at a level of detail not reached by conventional development techniques until test-planning. [24] I certainly agree. In SREM, the distinction between requirements and high level design seems unclear. SREM resembles a design language. Requirements in SREM are defined in terms of networks of processing steps in much the same way as a system design.

7.5. EDDA

EDDA, from Austria, is a data flow language with theoretical foundations in the mathematical models of Petri-nets. Petri-nets are pure mathematical models with a graphical form. A Petri-net consists of a set of places and transitions which are connected by directed vertices. Transitions have input places and output places. Tokens can be assigned to places in the net. If all the input places for a particular transition contain tokens, the transition is enabled and can fire. Firing means that the transition's input places become empty and the output places are marked with tokens. The Petri-net concept has

been extended by identifying tokens through data names, and by introducing firing conditions through guards. [25]

EDDA's graphical interface (G-EDDA) uses boxes for processes and arrows for data; it is nearly identical to SADT. [26] It is very understandable and therefore maintainable.

Graphs are translated manually into the specification language S-EDDA (Symbolic EDDA). The translation process is made routine by a one-to-one correspondence between the graphic and symbolic forms, with corresponding syntax and semantics. The formal semantics permit executable code to be generated by a compiler for S-EDDA programs.

The S-EDDA program for a G-EDDA activity defines the interfaces, types, data, data structures, subprocess interfaces, and the processing. Computer supported analysis checks for both static and dynamic semantic correctness. Another EDDA
feature is checking on timing behavior. This is possible because time requirements are stated for each activity, starting with the top level.

EDDA enforces hierarchical decomposition through stepwise refinement. It allows both top-down and bottom-up design. Existing subsystems may be inserted in new designs.

"G-EDDA programs [are] most useful as a basis for manual/automatic allocation of code to distributed processors." [27] Because EDDA is non-procedural, it permits parallel flow of processes. All that is required is the presence of data input at activities.

7.6. PSL

PSL is the "problem statement language" of PSL/PSA, a computer aided system for requirements documentation. Almost all of the system is written in FORTRAN. [28] The PSL/PSA technique consists of:

1. Recording the results of each activity in the system development process in computer processable form as it is produced.
2. Maintaining a computerized data base containing all the basic data about the system under development.
3. Using tools to produce hard copy documentation from the data base.

The PSL system description language is relational. Descriptions consist

of identifying and naming objects and relationships among them. The language consists of keywords for the type of information being documented. The objective is to express system documentation in syntactically analyzable form. PSL contains types of objects and relationships that permit descriptions of input/output flow, system structure, data structure (relations), data derivations (internal), system size and volume, system dynamic behavior, system properties (data objects), and project management. [29]

Procedures are natural language high level statements; e.g. "Compute gross pay for time card data." The designers of the language purposely omitted any procedural code so that analysts would concentrate on requirements rather than low level details. However, "PSL must be extended to include more precise statements about logical and procedural information." [30]

System information expressed in PSL is entered into a data base using PSA (Problem Statement Analyzer). Besides maintaining the data base, this software package produces reports to aid requirements analysis. PSA produces a system definition report containing the system requirements plus:

1. Narrative information as necessary for readability. This is stored in the data base and displayed with the system description, but it is not analyzed by PSA.
2. Lists, tables, arrays, and matrices prepared from the data base.
3. Diagrams and charts showing the relationships between objects.

PSA produces several other types of reports. A data base modification report provides a record of changes to the PSA data base. Reference reports include a Name List Report of all objects and types; a Formatted Problem Statement report that documents the properties and relations of each object; a Data Dictionary Report. Summary reports include a Data Base Summary that provides management information; a Structure Report that shows hierarchies; an Extended Picture Report that shows data flows in graphical form. Analysis reports include the Contents Comparison report showing similarities of inputs and outputs; the Data Process Interaction Report is used to detect gaps in the information flow or unused data objects; the Process Chain Report shows the dynamic behavior of the system. [31]

7.7. IORL

IORL is the formal language for Teledyne Brown Engineering's TAGS methodology. TAGS is an acronym for Technology for the Automated Generation of Systems. TAGS is composed of the Input/Output Requirements Language (IORL), the TAGS tool system, and the TAGS methodology.

IORL is a graphics and tabular language used to specify data flow, control flow, and detailed logic for the system to be developed. The system is described in three series of hierarchical diagrams. The highest level is the Schematic Block Diagram (SBD), which identifies all the principal system components and the data interfaces that connect them. The SBD components are decomposed in lower level block diagrams until the resulting components can no longer be subdivided. The second set of diagrams is the IORTD, Input/Output Relationships and Timing Diagram. These show the overall control flow for a single SBD component. Third, the PPD, Predefined-Process Diagrams, depict the detailed logic flow for a single process referenced in an IORTD or another PPD. The PPD's structure is similar to an IORTD. PPDs are used to improve readability of the specification, identify component dependencies and present the specification in a hierarchical manner. [32] Data is shown in tabular format in the Input/Output Parameter Table (IOPT) and an Internal Parameter Table (IPT). The IOPT shows data that passes over an interface between two components and defines the data for both components. The IPT defines variables that are internal to one IORTD and its associated PPDs. [33]

The IORL system definition is entered into a data base and manipulated by a set of tools. A package called Storage and Retrieval is used for database access. The Diagnostic Analyzer is used to find errors in syntax and semantics and also static design errors. It can find over 200 types of static errors.

The designer uses the Simulation Compiler to generate a definition of run-time parameters, simulate the system created in IORL, and process the input data to produce output. The Simulation Compiler checks for dynamic errors; these can be corrected using the Storage and Retrieval package. The combination of static and dynamic checking improves design quality. The use of simulation permits algorithms to be tested and alternative designs to be compared. [34]

"The specification step is an attempt to develop a prototype of the

system, a prototype in which components are allocated to an implementable architecture. If the prototype can be verified as meeting all the system requirements, it will become the implementation." [35] Multiple prototypes may be developed until one is chosen for implementation.

Until a translator is developed, the validated IORL design is manually translated into the implementation language by non-programming staff. Testing follows to determine that the translation is correct.

7.8. VIENNA DEVELOPMENT METHOD

The Vienna Development Method (VDM), from the IBM Vienna Research Laboratories, is a general-purpose method based upon denotational semantics. The method was developed to cope with the complexity of the Ada language. Since the Department of Defense does not accept subsets as a solution to the complexity problem, IBM decided to use a systematic notation for developing Ada software which would lead to a disciplined use of Ada features.

A VDM system description is composed of data structures, which constitute the internal state of the system, definitions of operations which are used to manipulate the state, and restrictions on the state. Using the notation of predicate logic, the designer expresses the restrictions by formulating predicates, or data type invariants, that must always be true.

VDM defines data using a small number of primitive types and user-defined scalar types and ranges. The user can construct structured types for objects such as sets, lists, records, and mappings

The language does not currently support programming involving multiple tasks, but is being extended to handle concurrency. [36]

The method requires several steps of refinement through increasingly detailed levels of design until the design is detailed enough for implementation. At each stage, the design must be verified by formulating functions to show that every value of an abstract type (set or mapping) can be represented in the newly refined design structure. These functions also show that the refined operations correctly model the effects of the abstract operations. [37]

Use of Ada generic packages to implement the basic VDM data types and structuring mechanisms would eliminate the need for intermediate refinements and proofs. If the generic package is provably correct, the instantiations

would not require proof. [38]

7.9. EDE

EDE, developed in Germany, is a formal language for specifying embedded systems requirements. The language provides constructs to model a system's static and dynamic properties, constraints, and semantics. It describes a system's functions, parallelism and timing through constructs for functions, processors and signals. The concept of a signal provides a primitive for synchronous message passing as a means of communication between processors. A signal is an entry point to a processor that is otherwise closed. [39]

The EDE approach to system description views the system as composed of real world processes and an embedded system which controls those processes. The real world processes are described and the embedded system is specified by defining an abstract system model incorporating both system parts. The requirements therefore include all important aspects of the system.

The system is described in EDE in three parts. First, the static, unchanging system properties are modelled by defining domains of values. The values in the domain may be simple, such as integers or Booleans. A domain may include countable values with no interesting properties that are represented to EDE, e.g., TOKEN. Domains of structured values may be constructed. The designer can also formulate "well formed" constraints, which restrict the domains to a desired range of values. [40]

Next, the dynamic properties are modelled as classes (domains) of behaviors. The designer uses the constructors for static properties plus three constructors for dynamic properties only: domain of functions, domain of processors, and domain of signals. A domain of functions describes all the functions that map arguments of one domain onto another. A domain of processors describes all the processors that offer the same communication interface (signals) and that are parameterized by the same domain defined in the static system structure. A domain of signals describes a specific communication component of an interface. A signal domain determines the type of messages and the direction of the information flow. Wellformedness constraints can also be formulated for the dynamic properties of the system using invariants.

Third, the semantics of the system is given by defining function objects

or processor objects, which represent a single object in a previously defined domain. Functions are defined using a subset of VDM. [41] A processor is defined using expressions dealing with internal parallelism, timing, and sending/receiving signals. The last part of the system specification describes the configuration and the initial conditions. Constructs are available here to describe the connections via signals among the processors. [42]

Simulation is used to validate the external consistency of the design. An interpreter is being developed which maps EDE text onto the related Predicate Transition system. A simulator takes the generated Predicate Transition system and simulates the functional and timing behavior of the designed system. The simulation provides a validation of the external consistency of the system because the client can experiment with the specification in order to judge its usefulness for the application. [43] As implementation progresses, implemented parts can be executed with the simulated parts of the system. The use of simulation and invariants permits timing problems to be detected at an early stage in system development.

7.10. PLACES

PLACES (Programming Language and Construct Evaluation System), from the University of Maryland, is a different approach to a specification language. Rather than develop a new design language, the authors extended an existing language, PL/I, so that the new features could be incorporated into existing systems. PLACES has been implemented by adding a macro processor to the University of Maryland's PLUM PL/I compiler. [44] PLUM is a large subset of PL/I; its restrictions of PL/I are in features to enforce better programming practices. [45] PLACES provides extensive debugging facilities to check conditions at compile time and during execution. Furthermore, it extends PL/I with data abstractions and program validation statements.

Data abstractions are encapsulated data types consisting of a set of values and a set of operations. Encapsulated data types are defined in PLACES by an abstraction module, which consists of 4 sections: representation, initialization, exceptions, operations.

Program validation is provided through the ASSERT statement. ASSERT

causes checking of a condition at compile time, or at run time if the program verifier can't verify the condition when compiling. ASSERTs are put into a library. The source program gets them from the library and makes them available to the verifier.

The PLACES model "is not restricted solely to PL/I. . . A similar description of a data abstraction has been described for Pascal, and Ada has essentially all the required primitives to implement a similar structure." [46]

7.11. RLP

RLP is the Requirements Language Processor, part of GTE Laboratories' Requirements Processing System (RPS). The purpose of RPS is to define requirements that are consistent, unambiguous, nonredundant, and machine processable.

RLP is a table-driven compiler which accepts as input several application-specific requirements languages. This flexibility is provided by the compiler's use of language definition tables. [47]

RLP modularizes the requirements in separately specified "features," facilitating multiple authorship. It formats the requirements document for readability and produces a table of contents and cross-reference indices. [48]

After checking for incompleteness, inconsistency, ambiguity, and redundance, RLP produces a machine readable finite state machine model of the system. The FSM description can be used by design, implementation and testing tools.

The FSM model is input to a Feature Simulator which enables execution of the specification through a prototype. This allows the client to interact with the system before design or implementation and to verify that the system is the one that is desired. Inputs to the simulated system are indicated by typing commands. System responses are indicated as messages to the terminal.

In the design phase, requirements are decomposed into smaller, less complex components by a tool such as PSL/PSA. RLP permits the designer to include previously defined system requirements in component designs and to enter new requirements for a particular component when necessary. RLP is used in design to define the external behavior of components. Once the external behavior of all components on a given level are specified, the Software Performance Simulator (SPS) is used to analyze the efficiency of the design. The SPS

analyzes the FSM model of each architectural component and during simulation calculates additional information concerning the probability and distribution of the various inputs. It can then predict system efficiency and make recommendations concerning the need for optimization or redundancy. [49]

For the implementation phase, the Automatic System Implementor generates the implementation from the requirements using the FSM and the Feature Simulator. It uses the results of the SPS to transform the architecture to the most efficient architecture. [50]

For testing, the Test Plan Generator (TPG) produces a set of certification tests for the FSM model. These are test plans for the completed system. Another tool, the Automatic Test Executor (ATE) provides inputs to the system according to the instructions of the test plan. The ATE reports on the success or failure of each test. "Neither a person nor a program can generate a complete set of tests. However, the RLP-TPG team can make the task of generating requirements tests more manageable, reliable, error-free, nonredundant and far less costly." [51]

For maintenance, the Feature Simulator is used to enact events in order to determine whether a requested maintenance item is a new system feature or a correction for a system defect. If the simulator does not provide a feature, it is a new requirement, not a system correction. New features are added first to the requirements so they can be simulated, then the design is updated by automatically decomposing the new requirements document using the decomposition tool. A new system is generated by the Automatic System Implementor. The Test Plan Generator uses the updated requirements model to derive new test plans for execution by the Automatic Test Executor.

7.12. SDL

SDL (System Definition Language) was developed at the University of Michigan. The language was designed to provide capability for simulation and to be compatible with PSL. This compatibility permits the use of PSL's static analysis tools. SDL is intended as a replacement for PSL in situations where simulation is necessary.

The model for SDL is relational. The entities are modelled as OBJECT types, the relationships as RELATIONS, and the attributes as PROPERTIES. SDL

retains the PSL form for OBJECTS and RELATIONS for static aspects of the system: system input/output flow, system structure, data definition and structure, data derivation and manipulation, static analysis and project communication. [52] It replaces the two PSL aspects, system size and volume, and dynamics, with three aspects: system modeling, resource management, and system dynamics and control. SDL uses a process interaction approach to modeling; the analysts views the system as a set of interacting activities.

The System Definition Manager enters the SDL system definition into the System Description Data Base and provides static analysis and non-quantitative dynamic analysis. [53]

Using the definition in the data base, the SIMSCRIPT Model Generator produces source code for a simulation language compiler, which creates a simulator for the system under development. SIMSCRIPT is the simulation language. The SDL forms for system modeling, resource management, and system dynamics and control resemble counterparts in SIMSCRIPT. [54]

Maurel and Bonnet point out drawbacks to SDL which I interpret as applying to PSL as well. [55] Ambiguities and incomplete specifications are generated in SDL because of the lack of semantic descriptions of the actions. A possible result is discontinuity between the specifications and the system in later phases. In addition, SDL lacks the data representation techniques essential for describing virtual devices.

SECTION 8. ADA AS A SPECIFICATION LANGUAGE

General specification languages do not map well to Ada. Either the specification language does not fully exploit Ada's features or not all features of the specification language can be used with Ada. Or else treatment of common features such as tasking, or typing, may be different in Ada and the specification language. [1] A specification language that is based on Ada would eliminate these problems.

Using the same language for both coding and design permits system evolution using consistent tools and notations. [2] The single language approach promotes construction and modification of prototypes. It eliminates the problem of information loss or distortion in the transition from design to implementation and it also facilitates that transition. It promotes component reuse because of ease in identifying matches between design components and existing modules. Other advantages of using an implementation language for design are described by Sammet, Waugh and Reiter. [3] Maintenance of the design can easily be part of maintaining the actual program. Metrics for design are closely related to the code, so that the design may be measured, not just the target code. Configuration control for the target language can be applied to the design as well. Learning the design language is part of learning the target language and conversely; the learning process is mutually reinforcing. There is less need to invent a design notation; the notation is already present in the target language.

The advantages of using a subset of the target language as a PDL outweigh the disadvantages. (1) There is always the tendency to code and not to design. Too much detail may be given too soon, constraining the implementation. (2) The rigid syntax of the language may inhibit the designer from thinking purely about the design. (3) No changes to the design language are permitted. There would be a disadvantage across projects when some aspect of the design language is changed. [4]

Even if the same language is not used for design and implementation, Ada recommends itself as a design language. Conversion of a design to the target language is easier to do with a structured language like Ada. "The conversion

process can be automated by approximately 75% (depending on the application) when moving to a FORTRAN or Pascal implementation." [5] Corliss names Ada tasks as the "most natural way to express . . . modules with continuous monitoring functions." Even if implemented in a language other than Ada, "the use of parallel tasks simplified the design by isolating unrelated functions and by allowing inter-task communications to be implemented as rendezvous." [6]

Ada is especially suitable for design because it provides a wide range of expression for problems. Because of its readability, it "captures the design in the software itself," limiting the need for external design documentation. [7] The language adapts to solutions rather than forcing the designer to fit solutions to the language. Ada provides tools for expressing abstract objects and operations; it is extensible, so that the designer can build additional problem-specific abstract objects and operations. Furthermore, Ada enforces the logical properties of those abstractions. Ada supports information hiding and other modern programming practices. Finally, the capabilities of Ada permit breaking away from the imperative sequential mind-set and thinking in terms of the problem space. [8]

Hart lists Ada features to aid software design: [9]

1. Structuring of the software architecture through packages and the visibility mechanisms. Packages are Ada's "single most important contribution to managing . . . complexity." [10]
2. Decomposition into modules (separate work units) with separate compilation. A procedure can be given a descriptive name that communicates its function. The details can be deferred for later implementation.
3. Specification and control of interfaces (subprogram declarations and package specifications) while deferring or hiding implementation details.
4. Tailoring the design to the application through appropriate choice of names and through definition of "global data structure templates" (types). According to Maurel and Bonnet, an abstract data type provides a unified, consistent data structure [11].
5. Declarations of global data structures through declarations and packages.

6. Depiction of logical relations between modules through subprogram calls.
7. Depiction of high level sequencing logic through sequential control structures.
8. Use of unstructured English to specify processing for which detail is deferred. This is provided by Ada comments, which should be preserved in the completed code as part of the documentation.

Additional Ada features for design activities are:

9. Exceptions. These "provide a syntax for handling abnormal conditions without adding additional complexity to the design." [12]
10. Generic subprograms, which allow the designer to specify programs independent of the data types to be used. [13]
11. The case statement provides better readability for abstractions in specifications. [14]

External circumstances also recommend Ada as a design language. It is well defined, supported by a large organization, and will probably become a common reference for the designers and programmers of large systems. [15]

Using Ada, the approach to design is to compile the specification. Unknown elements are made private with a minimal temporary completion so that they will compile. Only the minimal outside view of an object is specified for high level design. Package specifications are used to capture the intent of the design without binding any implementation decisions. [16] Compilation detects interface problems so that they can be corrected during the design phase.

The fact that Ada is a programming language causes problems when Ada is used for design. "Most programming languages are concerned with the expression of algorithms whereas program designs need primarily to express data flows and processing requirements." [17] An important deficiency of Ada is that it "provides no way, aside from informal comments, of communicating package functionality." [18] The meaning of references to a package are hidden in the package body. Besides semantics, Ada is unable to express assertions. Alstad discusses the inability of Ada to express assertions about the state of a computation, about the state of the external world, and about the effect of a processing segment at a lower level (i.e. requirements at a low level). [19] Goldsack discusses three weaknesses of Ada. It lacks primitive types commonly needed

for specification activities: iterated types, union types, and set types; it lacks concepts for relationships between objects; and it lacks broadcast messages for distributed systems. [20] Because of these deficiencies, specification languages have been developed to extend the Ada language.

8.1. ADA-RELATED SPECIFICATION LANGUAGES

Many specification languages are variations on the Ada language. Ada is used because of its advantages for the entire life-cycle and because many of the required facilities of a PDL are provided by Ada and are easy to use. [21] The following languages are based upon Ada but adapt the language for the specific purpose of expressing system design.

8.1.1. ANNA

Anna (Annotated Ada) was developed at Stanford University. [22] Anna extends Ada through (1) generalization of constructs already in Ada; (2) addition of new kinds of constructs, mostly declarative; and (3) addition of new specification constructs, mainly to specify packages and composite types. [23] Anna provides annotations which can be used to explain program behavior. It provides an axiomatic semantics that "can be applied to verify Ada programs by mathematical proof of consistency between Ada text and its formal Anna specification." [24]

An Anna program is an Ada program with formal comments defined by Anna syntactic and semantic rules. There are two kinds of formal comments in Anna, virtual text and annotations.

Virtual Ada text is Ada text that is marked as a comment with a virtual comment indicator (--:). This type of comment is used to define programming concepts (through mathematical or Boolean-valued functions) that are not included in the implementation. A virtual concept can be defined either by annotations or by a virtual body. If a virtual body is given, the concept can be compiled and executed to provide a basis for testing and validation. A virtual comment can compute values that are not computed by the program but which are useful in explaining what the program does; e.g., a history sequence of values of an actual variable in the program. A virtual comment must be

legal Ada. It must not influence the computation of the underlying Ada program by changing the value of actual objects; this keeps the program consistent with the Anna specification. And it must not hide entities in the actual Ada text by having the same name.

Annotations are built up from Boolean-valued expressions and reserved words indicating the kind and meaning of the annotation. Anna provides different kinds of annotations, each associated with an Ada construct and introduced by the reserved word --|. There are annotations for objects, types or subtypes, statements, subprograms, packages, exceptions, and context. Most annotations are constraints on values over their scope. [25]

Object annotations constrain the values of program variables within a declarative region.

Type and subtype annotations follow the declaration and are introduced by the reserved word --|where. E.g., --|where X:5 => C(X) and C(X) is a Boolean expression. Values of the type must satisfy the constraint of the Boolean expression. E.g., --|where X:EVEN => X MOD 2 = 0. "Subtype annotations generalize the Ada range constraint and can express more subtle properties since the constraint can be any Boolean expression." [26]

Statement annotations constrain the state after execution of the statement or constrain the execution of a compound statement. These are used to express simple kinds of program specifications such as assertions and loop invariants. [27]

Subprogram annotations constrain all calls and also the declarative region of the subprogram body, and place constraints on formal parameters and results of function calls.

Exception propagation annotations specify conditions under which an exception may be propagated.

Context annotations specify a list of variables from outside which may be used with the following unit.

For packages, visible annotations specify the visible part of the package, its data types and subprograms, so the user can understand how to use the package. These can introduce additional properties of the implementation that would not be obvious from the visible part [28]; they provide further specifications for the implementation. Hidden annotations specify the intended behavior of the hidden part of the package, the private part and body. They

may also define virtual functions declared in the visible part in terms of local items in the private part and body. [29] Annotations express package states, the values of variables after sequences of subprogram calls; these constrain the implementation. Annotations also express package axioms, algebraic relationships between package subprograms, by relating different successor states resulting from sequences of package operations. "Axioms are visible promises that may be assumed wherever the package specification is visible, and they are constraints on the hidden part of the package." [30] Annotations also provide a means of specifying the properties of a program's underlying domain of values. [31]

Anna provides quantified expressions that extend Ada expressions with the two quantifiers: (1) for all, and (2) exist. For example: for all X: DAY => exist P: PERSON_RECORD => P.BIRTHDATE.DAY = X means that "for all values of X of type DAY, there exists a variable of type PERSON_RECORD such that the component DAY has the value X." [32]

Anna specifications can be transformed into an Ada program. The annotations provide the basis for run-time checks for consistency with the original annotations and automatic reporting of inconsistencies. [33] A preprocessor to the Ada compiler transforms annotations into equivalent sets of simpler annotations that finally reduce to assertions. An assertion is translated into Ada text that checks whether the assertion is satisfied by a program state. Checking code is compiled and executed with the underlying Ada program. Exceptions are raised automatically at points of inconsistency. [34]

Anna does not provide the capability of expressing concurrent systems. "Special facilities for tasking are not included, hence the subject of specification of concurrent computation is still very much a matter of research." [35]

8.1.2. DAD

Dad was developed by Serge Savoysky of Laboratoire Central des Ponts et Chaussees, Paris. [36] The language extends Ada using category theory as a vehicle to formally describe the ideas of specification. [37] Dad provides a set of new language elements for states (data flows) and machines, and a set of laws for their behavior.

Dad models system elements as categories or functors. A category is an

abstract type. A category is a structure that models all the sets of functions for sets of values or states of a system element. A functor is also called a flux. Functors are associated with tasks; they represent machines performing functions on other elements. [38] When data is declared, it is associated with a category or a flux. [39]

The most general expression of machines in Dad is as a process. [40] A process defines a correspondence between its inputs and outputs. It has a visible part and an internal part. A process may be part of a larger structure: a system, device or component. Behavior is described by structured sets of expressions for possible ways of functioning, known as actions. [41] Behaviors are sequences of groups of actions ordered in time. Groups are sets of concurrent actions. [42]

Dad includes representation of time through the COMMAND attribute, which localizes an element in time. Time localization is used to synchronize exchanges between elements of the design. [43]

8.1.3. ADL

Ada Design Language (ADL), was developed at Ford Aerospace and Communications for detailed software design specifications. [44] It uses a large subset of Ada data types and control structures . It differs from Ada in three significant ways. [45]

1. Certain Ada constructs may be left incomplete.
2. It provides "To Be Done" constructs (TBD).
3. It has a prototype library of commonly used packages.

ADL requires all procedures and function calls, along with passed parameters, to be written in compilable Ada code. This permits verification of detailed design interfaces.

To implement the TBD construct, ADL defines a special package, PACKAGE_TBD, with types, records and arrays defined as TBD. In addition, a procedure, CALL_TBD, is defined; this has no parameters and performs no actions. The compiler will then accept TBD elements in the design. This permits unknown items to be left unspecified until they are firmly defined. It eliminates the need to recompile whenever an interface changes or when a type is changed. [46]

ADL uses comment statements containing structured English text. These are used for detailed processing requirements in a procedure or function. The use of structured English allows independent coding of the modules in various languages, not necessarily Ada. [47]

Data dictionary packages are provided by ADL. These allow a designer to refer to data in the dictionary without redefining data types and values within each package that uses the data. [48]

The only tool provided by ADL is an interpreter which performs syntactic and semantic checks of the ADL source code. The interpreter acts as a front end to the Ada compiler. It translates Ada into the DIANA intermediate form. [49]

8.1.4. PDL-ARCTURUS

PDL-Arcturus was developed by University of California at Irvine. Arcturus is an environment that includes the program design language PDL-Arcturus and also support tools for pretty-printing, directory listings, editing, performance monitoring. It provides a compiler (for optimum performance) as well as an interpreter (for error-detection).

PDL-Arcturus "uses normal Ada syntax forms in which the designer substitutes text in braces ({}) in place of declarations, expressions, names, statements, or types." [50] Subprograms and packages with these {comments} can be executed in Arcturus. Ada statements are executed, but Arcturus executes a "break" package when it encounters a {comment}. If the {comment} is not a statement, program execution is halted.

The Arcturus environment also provides a "Rapid Prototyping Language." This enables the designer to define a macro that generates expanded code for an Arcturus comment. The result of the macro execution replaces the "calling form" (the {comment}), so that the macro is executed and replaced only once during program execution. This is referred to as the "calling form macro facility."

PDL-Arcturus promotes reusable modules by permitting association of a coded implementation with a design construct through the calling form macro facility. The macro allows the designer to determine if a past Arcturus {comment} matches the {comment} that she/he wishes to use. If there is a

match, then the {comment} and macro can be reused for the new purpose. As a result, there is less likelihood of multiple {comments} that perform an identical function. The developers of PDL-Arcturus cite a case in which 62% of a prototype was built with reuse software. [51]

8.1.5. ADA-PDL

TRW's Ada-PDL uses a relaxed Ada syntax and expanded data types. It permits free form English in Ada statements to aid design at all levels. A comprehensive set of tools is provided with the language.

Ada-PDL includes formal and informal constructs. The formal constructs extend Ada with facilities useful for design. They must be written with a specific syntax similar to Ada's. The informal constructs are almost free of syntactic constraints. These include design narratives, which follow an Ada keyword, and comment constructs, which begin with -- and can appear in algorithms or name declarations. [52]

The basic Ada-PDL statements are free-form beyond the leading keyword or the declared name. [53] Certain elements of the design are permitted to be absent, for deferred refinement. The language also collapses Ada's syntax for defining records and enumeration types into direct data declarations. It expands Ada's built-in data types to include other well-understood structures. [54]

The tool set includes a cross-reference list, a name directory, as well as reports of module dependency, call hierarchy, and parameter checking. [55]

Hart recommends Ada-PDL for retraining software developers in the new features and complexities of Ada. Its syntactic simplicity permits focus on design issues rather than syntax, and it makes it readily acceptable because it is easy to use. Rather than immediately converting to use of full Ada for design, using Ada-PDL enables a gradual evolution from prevailing software practices. The Ada-PDL processor can be easily integrated with other full-APSE tools. Of most significance is that Ada-PDL's lack of low-level Ada constructs and its appropriateness for design discourage coding during the design phase. [56]

8.1.6. PDL/ADA

IBM Federal Systems Division's Ada/PDL has been used with Jovial and PL/I as target languages as well as Ada. [57] PDL/Ada maps PDL to a proper subset of Ada. The emphasis is on keeping PDL/Ada as small as PDL in order to prevent coding rather than design. [58]

Because PDL/Ada is a subset of Ada, it is acceptable to the Ada compiler and all tools that accept Ada. Although the compiler will not generate code for PDL/Ada designs, it analyzes syntax, checks for closure of the language structures and performs type checking. [59] The subset excludes from the Ada formal grammar any productions that are not applicable for design. [60]

In order to express high level concepts in informal language, the language provides a Boolean variable CONDITION and a null procedure, THENPART, to use as components of an IF statement. For each IF statement, the meanings of CONDITION and THENPART are provided by comments. [61] Use of the comments alone, without the specially defines predicate and procedure, would not constitute an acceptable IF statement.

PDL/Ada provides the following Ada features. [62]

1. Data. Primitive types, enumerated types, constants and variables, arrays and records, user-defined data types. Pre-defined types such as stacks, queues, sets and sequences are pre-defined as Ada packages. Another package provides definitions and operations for handling character strings of variable length.
2. Statements. Assignment and procedure call. Procedures are called by writing the procedure name.
3. Control structures. Sequencing, looping, and branching. Loop constructs include while do, do until, and do while do. Branching includes if then, if then else, and case.
4. Components. Functions and procedures with both positional and named notation, packages. Generic packages are included because they are needed for defining abstract data types. However, generic programs are excluded because PDL (which PDL/Ada maps to Ada) does not contain generics. PDL/Ada includes the with, use, and separate clauses so that design components may be created separately.

PDL/Ada does not include exceptions, tasking, nesting of procedures, initiali-

zation of data, derived types, overloading of operators, full generics, and the GOTO statement. Exceptions and tasking may be included in later versions of the language. [63]

8.1.7. BYRON

Byron, a product of Intermetrics, Inc., adds constructs to Ada programs as formal Ada comments. The Byron processor uses the comments to generate design documents and to perform some design analysis.

Byron includes two types of design constructs, directives and flags. Directives are introduced by -- followed by a keyword and text. Several keywords are provided, to express data abstraction, program description, timing requirements, exception handling, and performance analysis. The text part of the directive describes the concept indicated by the keyword.

Flags are introduced by ---. They are used to denote the scope of Byron statements. They are used to mark the beginning and the end of a block to be processed by Byron. Byron also provides other characters following the Byron prefix -- that are used for a similar purpose.

Several tools support the Byron language. An analyzer performs checking for correct Ada syntax and semantics. A calling tree tool reports the functions and procedures that a program unit calls as well as the functions and procedures that call it.
A data dictionary displays declarations in a selected set of program units. A dependency table tool reports dependencies among program units, showing which units must be recompiled if a given unit is modified. A user manual tool creates a report describing the external interface to a unit in the program library. [64]

8.1.8. ADLE

Ford Aerospace's design language ADLE (Ada Design Language Extensions) extends Ada, Anna and ADL with set theory constructs and annotations for tasking.

The syntax and semantics for set theory are borrowed from HDM. The HDM semantics are modified to be consistent with Ada's type checking. [65]

Keywords provided for sets are SET_OF, UNION, INTER, DIFF, SUBSET, INSET, and CARDINALITY.

The annotations for tasking are used to indicate: (1) the necessary conditions for a rendezvous to occur; (2) exceptions that may be propagated within a rendezvous or within a task; (3) state changes in a task or state changes resulting from a rendezvous. [66]

8.1.9. ADA/SDP

ADA/SDP is the programming design language for Mayda Software Engineering's System Design Processor. Developed in Israel, it has been used in Israel, France and the U.S. Mayda's "approach uses a simplified Ada syntax with features that facilitate a more natural and readable design description." [67] ADA/SDP includes the language and a processor to provide automated analysis and documentation aids.

The language uses pseudo-code that combines natural language expression with Ada control structures and declarations. The goal of the language is to permit design solutions to be expressed as they evolve. It includes all the Ada features that contribute to design, along with PDL support for abstract, incomplete ideas that often can be expressed only in natural language. The Ada syntax is relaxed so that the restrictions, which do not support the design function, do not distract and therefore inhibit the design process.

Design descriptions are made up of modules which may be subprograms, package specifications or bodies, or task bodies. The modules are maintained in libraries of separate design units. They are presented to the processor unnested, in top-down order as developed. A module is made up of one or more pseudocode declarations or statements. A statement may be an Ada statement, a reference, or just text. Text may refer to declared data objects or types. To encourage clarity in the design, identifiers and type declarations may be sentences. Formal parameters are incorporated into the module name. (For example, procedure PUSH VALUE INTO STACK has parameters VALUE and STACK.) The processor accepts full Ada, although Ada is not fully checked. [68] The processor checks misuse of declared items, type checks parameters in order to verify interfaces, identifies mixed types in text statements. It provides cross-references for subprograms, packages, tasks,

types, record components, data objects, and labels. It displays module designs in pretty print and has an option to generate Ada source code.

8.1.10. ADADL

ADADL stands for Ada-based System Design and Documentation Language. This product, from Software Systems Design, combines Ada data and program structures with an Ada based pseudo-code for processing design. The language provides a consistent expression through top level and detailed design; it includes full Ada. Designs in ADADL have been implemented in C and Jovial. [69]

Control flow and logic designs in ADADL are expressed in an Ada-based pseudocode. Ada is used for program and data definitions. The ADADL processor analyzes the design for errors and produces "custom reports to track such things as each program unit's requirements traceability, and the dates of completion of the design, coding, or testing of each program unit." [70]

ADADL includes more than 25 tools written in C. [71] These include reports of program structure, data declarations and use, type declarations and use, calling hierarchy, instantiations of generics, interrupt information, data dictionary. There is a tool that calculates the design complexity for programs and the system as a whole. A pretty printer formats the design and highlights the keywords and program invocations. A test generator designs strategies for unit testing and helps prepare design review material. For military contractors, there is a tool to automatically provide DOD standard documentation from the design. [72]

8.2. GRAPHIC REPRESENTATION

A graphic representation clearly communicates basic relationships between design units. Although, as of 1987, there was no clear consensus in IEEE concerning recommendations for graphic forms for expressing Ada design [73], some approaches use common pictorial elements.

Grady Booch uses a graph of the entire system to show visibility relationships among packages. This is the high level system graph. At a lower level, he provides diagrams of the external and internal views for each package. The external view depicts the types and operations in the package's interface, the

parameters, the packages that interface with the subject of the diagram, and the packages with which the subject interfaces. The internal view depicts data stores, significant internal procedures, and the relationships between the internal elements of the package. Examples of these types of graphs are included in Appendix G.

Booch's graphs resemble those of R.J.A. Buhr. Buhr makes extensive use of graphs to depict designs for example systems. He provides precise specifications for all elements of his pictorial notation. [74] Clouds or globs are used for uncommitted modules. Boxes are used for packages and tasks. The boxes for packages are rectangles for packages and parallelograms for tasks, representing the parallel nature of tasks. The entries of tasks and the data types and procedures forming the interfaces of packages are depicted as "sockets" that users may "plug into." Access connections to the sockets are indicated by arrows drawn from anywhere on the user box to the outside edge of the socket.

Buhr provides detailed notations for tasks. Multiple arrows may be numbered to indicate order of access to an entry. Or they may be marked by an arrow drawn across the access arrows in the direction of the access order. A set of entries accepted in time order is indicated by drawing a line around or across the set of entries. He uses a bent back arrow to depict conditional entry calls to tasks. A conditional entry call labelled with a T indicates a timed entry call. Dots are used to represent guards.

8.3. STYLE

Clarity must be the primary guideline for choices involving style. [75] Both the application of the language and the presentation of the design must promote readability and understandability. The design must be written "with the reader in mind, not the writer." [76] Guidelines for style should be chosen carefully and followed as closely as possible. "Consistent application of reasonable . . . conventions promotes a uniformity of notation that helps to simplify the work of the . . . reader." [77]

The style of the system design itself should abstract both data and control structures, hide unnecessary detail, limit the entities at a given abstraction level to a manageable number, exhibit strong cohesion and weak

coupling of program units, and reflect the real world of the problem. [78]

Grady Booch provides a style guide for using Ada to specify system design. It includes suggestions for types of applications that are appropriate for each of these Ada constructs: subprograms, packages, tasks, generic program units, exceptions.

The choice of descriptive names is fundamental to understandability. Booch's conventions for names include use of underscores in names for readability. Names should be long enough to describe the basic characteristics of the named component. Types are named as common nouns, e.g., TREE, LIST. Objects are named as proper nouns, e.g, MY_TREE, PERSONNEL_LIST. Packages that are not objects are named as noun phrases, e.g, MATH_FUNCTIONS. Procedures are named as active verbs, e.g. SORT_LIST; Boolean functions are forms of the verb to be, e.g. IS_NOT_EMPTY. Tasks are noun phrases denoting some action, e.g. TIMER, LIST_SEARCHER. [79] Exceptions are also expressed as proper nouns, e.g., DUPLICATE_RECORD.

The use of named parameter association is recommended for readability.

8.4. APPROACHES TO USING ADA FOR DESIGN

The literature describes a variety of approaches to using Ada for design.

8.4.1. SUBSET OF ADA

The purist's approach is to use only Ada. (Booch, Buhr, Maurel and Bonnet) Because the complete language is not necessary for design, a subset of language features is designated to be used. The basic Ada features for inclusion in the subset are: [80]

1. Libraries and separate compilation.
2. Packages, with separate specification and body, to allow separation of data and algorithms.
3. Application oriented packages.
4. Abstract data types - user defined types and object declarations.
5. Control structures - the complete set of Ada control structures.
6. Statements - assignments and subprogram calls.

7. Error handling - use of the exception handler section; may have comments for text.

The advantage of the pure subset approach is that a consistent syntax is used throughout system development; the design language is acceptable to any tool or program that supports Ada; the subset makes unavailable implementation-oriented features that are too specific for design. [81]

8.4.2. SUBSET PLUS DOCUMENTATION

IBM Federal Systems (Waugh) uses a subset of Ada augmented with detailed documentation. [82] The advantages of this approach are increased understandability and inclusion of information that is not directly expressible in Ada. Booch implies that extra documentation is unnecessary. He points out that Ada can be virtually self-documenting. [83]

8.4.3. RELAXED SYNTAX

Hart (Ada-PDL) and Yavne (Ada/SDP) advocate a freer form of Ada which relaxes some of the syntactic rules. Conversion of the design to proper Ada syntax occurs in the coding phase. Permitting relaxed syntax lessens the effort required to express the design, permits focus on the design rather than the language, and facilitates transition to Ada from other languages. A simpler design language can use simple processing tools that are less resource intensive.

8.4.4. CONTROLLED COMPILER INTERPRETATION

Anderson suggests using a pragma to control compiler interpretation of the language to permit design elements. The pragma enables the compiler to distinguish between a design and a final program; the pragma would be removed at implementation. Any design constructs would then become unacceptable to the compiler and would be flagged as errors. This would enforce completion of all parts of the design before the software could be released. Anderson's pragma is:

pragma DL [(options-list)] ; [84].

The pragma would facilitate step-wise refinement by enabling use of English narrative for unfinished design elements. The narrative could be contained in a To Be Determined (TBD) phrase denoted by curly brackets.

tbd_phrase ::= { English_text } [85]

The phrase must be properly nested in an Ada construct such as an if statement or a loop statement. Ada's BNF rules would be enlarged to permit the TBD phrase to be accepted in place of an identifier, numeric literal, declaration, type definition, expression, sequence of statements, actual parameter, or package specification.

8.4.5. TBD PACKAGE

Bardin (ADL) uses a "TBD" package containing place-holding constructs to be used when elements of the design are unknown. The package supplies type definitions, range limits, default values, and a procedure call. The TBD_Package is imported via the with and use clauses. This permits the designer to define initial abstractions, filling syntactic gaps with TBD constructs. For example: [86]

```
type Hidden is new TBD_Type;
Int  :  Integer := TBD;
type Static is range 0 .. TBD_Int;
```

The declarations can be refined in successive design iterations until all the dependencies upon the TBD_PACKAGE have been removed. Then the with and use clauses for the TBD package may be removed.

Like Anderson, Bardin suggests that eventually use of TBD constructs should be simplified by adding TBD to the language syntax, and using a pragma to turn the TBD feature on and off. [87]

Use of TBD constructs enables earlier execution to check the design consistency. It also "promotes integration of the PDL into a unified view of source text as a single, dynamically changing, multi-dimensional representation of the design." [88]

8.4.6. FORMAL COMMENTS

Languages such as Anna and RAMTEC's Ada PDL extend Ada through use of

formal comments. The comments begin with -- according to Ada syntax, followed by a keyword or symbol from the syntax of the design language.

Anna has two types of comments. Virtual comments define concepts that supplement the program but which are not explicitly implemented as part of the program. The text of the comment is legal Ada code that is usually a mathematical function. Annotations are used to provide constraints or to specify processing at a lower level. The syntax of the annotation depends upon the Ada construct that is annotated.

RAMTEC's Ada PDL requires formal comments at specified points in the program unit. Before the specification and body, a preface section is included as a formalized comment; it contains accountability, traceability, security, technical, and management information. Formal comments are used in the specification section to identify both calling and called program units. ("TBD" is acceptable.) Either the specification or body must contain a formal comment, --Functional description. The body may also contain a comment, --Operational description. [89]

At RAMTEC, the primary objective of providing the design extensions is to use software tools to process the superset and to generate either a detailed program template that can be completed during the coding phase, or else the completed program itself. RAMTEC reports large productivity gains as a result of applying such
tools. [90]

8.4.7. CONSTRUCT FOR FREE-FORM TEXT

Gabber's "middle way approach" between simplified syntax languages and full syntax languages is to add a new lexical element to Ada that permits free form text for loosely defined design elements. The element, called "escape," consists of any text enclosed in curly brackets:

{any free form text} = 'not defined yet'

It can replace almost any Ada syntactic entity except program structure keywords, program names, and parameter lists. [91]

Use of the escape construct is a compromise between two alternatives. Relaxed syntax permits ease of expression but disables thorough checking of the design by automatic processing. Full syntax can be automatically processed.

However, the design is more difficult to express; the design is not expressed naturally; and "the description of any missing part must be expressed as comments, detached from the body of the deign statements." [92] The advantage of the compromise is that it provides "one of the most important traits that a good PDL should have: [an] easy, natural way of expressing incomplete designs." [93]

8.4.8. SPECIFICATIONS PACKAGE

Pyle describes extending Ada with a SPECIFICATIONS package containing subpackages that provide notations for design in different degrees of formality.

There are three subpackages. PSEUDO_CODE is just that; it permits informal contents with a formal structure. It contains two procedures that accept character strings in quotes as parameters. The first procedure, ACT, is used to describe what an object does. The second procedure, IT_IS, is used to describe objects.

The second subpackage, OBJECTS, permits the use of specific identifiers. Its two generic procedures are ENTITY, for defining entity types, and ACTION, for defining procedures. Normal Ada rules apply when this package is used.

The third package, SEMANTICS, contains a procedure NOT_DESIGNED_YET, which may be used for package and procedure bodies. The procedure bodies may then be designed using the PSEUDO_CODE package.

The SEMANTICS package contains procedure ASSERT which is used to describe relations among variables that hold on entry to and exit from the procedure. To use ASSERT, the designer declares functions PRE_CONDITION and POST_CONDITION and procedure EFFECT in the design, right after the Ada specification of the unit's formal parameters. The functions are Boolean and return values true or false, depending on whether an expressed relation holds. Procedure EFFECT describes relations holding between entry and exit values of relevant variables, asserting the proper relationship. The notation for entry and exit values of the variable X is X'IN and X'OUT.

Pyle provides the following examples. [94] To express the post-condition that two variables are put into ascending order, the designer writes:

```
function POST_CONDITION return BOOLEAN is
     begin
          return A <= B;
     end POST_CONDITION;
```

To define the effect of swapping two variables, the designer writes:

```
procedure EFFECT is
begin
     ASSERT((X'IN = Y'OUT) and
       (X'OUT = Y'IN));
end EFFECT;
```

To express "universal and existential quantifiers" for scalar types, the designer must use Ada loops or generic units. For example, to say that all members of an array are made positive, where

```
type INDEX is range 1:10;
A: array(INDEX) of FLOAT;
```

the designer writes:

```
function POST_CONDITION return BOOLEAN is
begin
     for I in INDEX loop
       if not A(I) > 0.0 then
              return FALSE;
       end if;
     end loop;
     return TRUE;
end POST_CONDITION;
```

or, instantiating a generic package, the designer writes

```
function A_POSITIVE(I : INDEX) return BOOLEAN is
begin
     return A(I) > 0.0;
end A_POSITIVE;
function POST_CONDITION is new FOR_ALL(INDEX, A_POSITIVE);
```

The SPECIFICATIONS package was developed in an effort to use Ada as far as possible in design, so that existing Ada tools could be applied. Pyle concedes that the notations using the package "turn out to be rather clumsy and over-verbose." [95] The package does not address "performance (time and storage constraints) or specification of input/output including man-machine interaction." [96]

SECTION 9. PROJECT DESCRIPTION

The thesis project was the high-level design for an example system, READY FENCE!, in Ada. The purpose was to gain experience in expressing system design in a formal language, to analyze the experience, and to contrast the Ada design with the natural language design for the same system.

9.1. THE EXAMPLE SYSTEM

READY FENCE! is an interactive system for managing a fencing competition. It is a tool for registering fencers for up to four events, ranking the competitors, and assigning them to bouts in pools or elimination rounds. It prints score sheets and reports, tabulates results, and advances the status of the competition. READY FENCE! is an actual system, designed to meet a client's requirements, but not yet implemented.

Appendices A, B, and C provide a detailed description of READY FENCE! and are an introduction to the system design. Appendix A, System Overview, describes the system data files and functions. The overview narrates the general procedure for an operator to use the system to manage a fencing competition. Appendix B, Sample Screens, provides details of the user interface. The entire set of screens is the complete functional specification for the external system. The design of the system structure is based upon this specification of the interface. The implementors must refer to these screen specifications to determine coding details for input and output routines. A study of the screens reveals the system functions and error processing. Appendix C, Requirements Definition, provides general requirements for the system. Specific lower level details for each function are developed in the design.

9.2. THE NATURAL LANGUAGE DESIGN

Before undertaking the thesis, I had already completely designed the system informally, using natural language descriptions and a Pascal-like pseudocode. There are 6 files, 96 modules, 30 screens, 4 reports, and 2 scorekeepers' forms.

The system was designed as a hierarchy of processing functions. Appendix D, Structure Graphs for Natural Language Design, illustrates this procedural structure.

Appendix E contains the natural language design for the high level modules. Each module design includes the module name, purpose, parameters, global variables accessed, and detailed design in pseudocode. The section on module purpose fully details all processing required of the module. Where appropriate, the reason for specified processing is noted. The pseudocode in the detailed design section resolves any ambiguity in the language of the purpose section. The module design provides complete documentation of all processing details for user, designer, implementor and maintainer.

The entire natural language design is too lengthy to include. The designed system completely meets the client's requirements and is ready for implementation. The full design is traditional, containing a data dictionary, module descriptions, and layouts for the screens, reports and score sheets.

The data dictionary contains entries describing system constants, global variables, and files. Each file entry is divided into two parts: a description of the file itself, and a description of the records in the file. The file description includes the file's purpose, uses for each record type, how the file is loaded and saved, the operating system file identifier, and the number of records. For each record type, the description includes all variables with their class, length, purpose, and values.

9.3. THE ADA DESIGN

I have produced a high level Ada design for READY FENCE! using pure Ada in the style of Grady Booch. [1] I have not completely re-specified this large system. The high level design sufficiently illustrates the use of Ada as a design language and shows the effect of the Ada language upon the design concept.

9.3.1. OBJECT-ORIENTED VIEW OF THE DESIGN

Following Booch's steps for object oriented design [2], I first developed an informal view of READY FENCE!'s abstract world. I wrote a short definition of the problem, then noted the nouns and verbs in the definition. The nouns

identify the objects in the design. The verbs identify the operations upon the objects. The following is the problem definition, with the nouns in upper case and the verbs underlined.

> Permit an OPERATOR to select functions to conduct a COMPETITION. For each EVENT in the competition, input FENCERS and add to a REGISTRATION list. Rank the fencers in the list, assign them to bouts in POOLS or ELIMINATION. Permit the operator to adjust the POOL TABLE of assignments to avoid matching fencers from the same fencing CLUB. Input results of previous rounds and update the statistics for each fencer. Maintain a TABLE OF FENCING CLUBS to which the competitors belong. Also maintain a SECURITY TABLE of operators permitted to use the system. Print SCORE SHEETS and REPORTS.

9.3.1.1. OBJECTS AND OPERATIONS

The above evaluation reveals the basic objects:

COMPETITION	ELIMINATION LIST	OPERATOR
EVENT	REGISTRATION	SECURITY TABLE
FENCER	CLUB	SCORE SHEETS
POOL	CLUB TABLE	REPORTS
POOL TABLE		

Appendix F lists the objects and operations for READY FENCE!. The operations include those underlined in the problem definition, plus some that were not expressed in the definition but which were readily apparent, and others for which the need became visible as the design progressed.

Getting this far in the design was not the simple process that I had expected from Booch's examples. Repeated effort was required to define the skeleton of the system in simple terms while including all the major functions. Once that was done, the objects were easily discerned, but not all the operations. The initial definition was useful primarily to reach a starting point for the design. The full list of operations did not spring from the initial definition but was obtained gradually during the later design stage of establishing the package interfaces. To include all the operations in the initial definition would have greatly complicated the definition and perhaps have made it more difficult to discern the objects.

9.3.1.2. PACKAGES

I had defined the problem, identified the objects and their attributes, and identified the operations on the objects. Next, I needed to determine the packages for the Ada system so that I could establish the visibility of each package in relation to the others.

To determine the packages, I classified each object according to Booch's list of resources that a package encapsulates: [3]

1. An abstract state machine.
2. A named collection of declarations.
3. An abstract data type.
4. A named collection of declarations.

This helped me to visualize how each package would be used and to determine what should be included in the package. The analysis focused upon the real world of the competition, upon the objects and the visibility relations among the objects, rather than upon the processing that would be required.

From the list of 13 objects, I chose 10 packages for the system. Closely related objects are combined in a single package in order to simplify the visibility. The final number of packages in the design is 11. For flexibility and ease of maintenance, I added a package of constant declarations and named it CONSTANTS. A description of the 10 principal packages follows.

(1) COMPETITION is the entity on the highest level. The competition is an abstract state machine (a data store) containing the state of the competition as a whole and the individual state of each event. The system operator can manipulate the state of the competition and can access the state of all the events. But the complete data for only one event can be manipulated at a time; this event is called the current event.

At the start of the session, we initialize the current event. This means that the operator views the current status of the competition and selects the event that is to be currently manipulated. At this time, the operator can close the existing competition and open a new one, or can open and close events. After intitialization, the operator may change from one event to another and back again using the change current event operation. The perform function operation manipulates the state of the competition by performing a function chosen from the READY FENCE! menu by the operator. The get report data operation

provides reports with the competition and event titles and all the event information needed to construct a report of the current event.

The other COMPETITION operations are obvious. The operator can open and close the competition. The system loads the competition at the start of the session and saves it at exit. The is open function informs the system whether the competition status is open or closed.

(2) EVENTS provides the abstract data type EVENT_TYPE and provides operations that COMPETITION applies to the selected current event, altering its state. We open and close an event; load and save the event data.

Before the competition, we build the registration list using the register fencers and the delete fencer operations. The order registration operation builds the name or rank index in preparation for reports and bout assignments. Assign pools and assign elimination build the pool table or elimination lists so that score sheets can be printed for the next round of competition. Review pools permits the operator to review and adjust the pool assignments made by the system. Input pool results and input elimination results input data for each competitor's performance in the previous round and update the fencer's status and accumulators. Once competition begins, a fencer is removed from the competition by the withdraw fencer operation. At any time, the correct registration operation is used to modify a fencer's data. Display elimination assignments and display pool operations do just that: display.

The get values operation passes the values of the event record in a separate data type so that they can be accessed for read only. Because EVENT-TYPE is an encapsulated data type, COMPETITION cannot access the individual items within the event record. Only EVENTS has this access. So EVENTS gives COMPETITION the read access in a separate unencapsulated data type, VALUE_TYPE. Ada's package construct and typing facility safeguard the event record from unauthorized manipulation and at the same time provide the ability to make the record visible. Typing enables privacy and visibility for that which is private!

(3) FENCERS makes available the system's principal object, fencer. Six of the nine remaining objects interact with the fencer object. Since each of them needs to access the individual fencer values, I made fencer an unencapsulated data type in its own package. The FENCERS package consists only of the declaration of the fencer type. Encapsulating the fencer type would have complicated using

the fencer object without preventing unauthorized objects from changing the values of the objects collected in the fencer object. I would have had to provide an update operation so that the fencer's accumulators could be modified according to results of each round. Any object could then use this operation. Booch provides this rule of thumb: "We tend to use encapsulation when it would be dangerous for a client to have access to the underlying representation of the entity." [4] Fencer is unencapsulated because other objects can't corrupt the system by having access to it.

(4) REGISTRATION is an abstract state machine. It stores and protects the fencer data for the current event, and provides operations for changing its state. Because the registration table is internal to the package, it can be altered only by the operations in REGISTRATION's visible interface. We need to load the registration table for an existing event, save the registration table after it is updated, and initialize the registration table when an event is opened. To build the table, we need the add fencer operation. We must be able to delete and modify fencers in the table. To prepare for access according to a key sequence, we will use build name index and build rank index. Then we can read by name or read by rank. The get fencer operation returns the fencer record for a given fencer ID.

(5) POOLS encapsulates the pool table data store, an array of POOL_TYPE, also encapsulated. Separate operations are provided to load, save, and initialize the pool table. Operations for the pool are to insert and remove a fencer from a given pool, determine the size of a pool, and get the ID's of the fencers assigned to a pool.

(6) ELIM_LISTS encapsulates the lists used for rounds of elimination. The elimination list is a list of fencer ID's in ordered pairs of opponents for bouts. Two other lists, the winner list and the loser list, record subsets of fencers from the elimination list that are used in assigning bouts for elimination with repechage. Again, we need to load, save, and initialize the lists. Other operations allow us to insert and remove a fencer from the elimination list, to record a winner in the winner list, record a loser in the loser list, and get a fencer from the elimination list, the winner list, or the loser list.

(7) CLUBS encapsulates the club table but provides an unencapsulated type, CLUB_TYPE. As in FENCERS, but on a smaller scale, other objects need the values of the objects assembled in CLUB_TYPE. We may allow access to the club

type, but we need to protect the club table, which contains index pointers. Operations are provided to load and save the club table, add a club to the table, delete and modify a club in the table, display and print the table. In club table is a function that returns a Boolean value for presence of a given club in the table. Two other useful operations are internal to the package and not contained in the package interface. These are search club names and search club codes; they find the location of a given club code or name in the index of club names or index of club codes. The add club and in club table operations use these internal operations.

(8) OPERATORS encapsulates the operator object type and the array of operators called the security table. There are operations to load and save the security table, add an operator to the table, modify or delete an operator. Validate operator accepts the user ID and the password given at entry to the system; it returns either the operator's security level or an indication that the operator and corresponding password are not in the security table. Operator inquiry and display security table list the data for a single operator or all the operators in the table. The operator password change allows the operator to change his/her own password at entry to the system.

(9) SCORE_SHEETS is a named collection of subprograms that print score sheets for a round of pools or a round of elimination.

(10) REPORTS is also a collection of subprograms that report the registration table. The highest level subprogram is specify report, which prompts the operator to select the type of report to be produced. The three available reports are in a similar format but ordered either by name, by rank or in the sequential order of storage in the table.

9.3.1.3. VISIBILITY

After identification of objects and operations, the next step in object-oriented design is to establish the visibility of each object in relation to the other objects. This entails determining the access relations between the packages and establishing the interfaces for each package.

I found the question of visibility a difficult one because it required grappling with unfamiliar issues. Many examples in the research had illustrated the use of private types in what seemed a very straight-forward manner. But these examples

were relatively simple and dealt with using an object as a whole, as in pushing on object onto a stack. They did not deal with a case in which the elements of a record type object needed to be accessible. However, Booch provided a model [5] in which a package provides values for an ecapsulated object. I used this technique for the EVENT limited private type as described above.

I also found that I could encapsulate an object by defining it within the package body. The object need not be a private data type unless other objects require access to it. The tables and lists are examples of such encapsulated objects.

The question of visibility was the principal focus at this stage of the design. But the issue had arisen in the earlier stage of determining operations. Some of the required operations for an object depended upon which other objects viewed it and what their needs were. And visibility would continue to be a central concern throughout the following stages of the design. For me, the visibility relations were the central consideration in the object-oriented design process.

9.3.2. GRAPHICAL REPRESENTATION

Appendix F contains 3 sets of graphical representations for the system.

The first is a single structure diagram illustrating the interconnectivity between the packages following the style of Grady Booch. The main procedure is the box with the line across it. As a subprogram, it is "represented by a linear structure, implying its sequential nature." [6] Each package is represented by a rectangle which symbolizes a wall surrounding the package contents. An arrow from inside package A to the outside of another package, B, indicates that package A uses a type or an operation in package B. That is, package B is visible and subordinate to the calling task, A. The curved lines and layout are meant to distinguish this graph from a traditional structure chart because "the topology of Ada systems is <u>not</u> strictly hierarchical." [7]

The second set of graphs consists of four Ada structure graphs in the style of R.J.A. Buhr. [8] I chose two of the more interesting packages that did not have an overly large number of operations, CLUBS and REGISTRATION. For each package, I diagrammed the external view, showing the interface with other packages and procedures, and the internal structure, showing the internal procedures and data stores. The operations and types in the interface appear at

the edge of the package wall and serve as sockets where other packages plug in to utilize the package resources. The socket for types is distinguished from an operation socket by its rounded corners. The parameters for each operation are indicated by numbers in the diagram. These correspond to numbers below the diagram labeling the parameter names and arrows that indicate whether the parameter is passed into or out of the package. The diagrams for packages with more operations and internal procedures would follow the same pattern as these two diagrams, but would be larger and more complex.

The third set of diagrams in Appendix F, covering 12 pages, is a traditional structure chart illustrating the calling hierarchy of the Ada READY FENCE! modules. I believe that such a chart is always useful to clearly reveal the designer's intent and also to provide documentation for reference in the system's design, implementation, and maintenance phases.

9.3.3. MODULE DESIGNS

Designs for the high level READY FENCE! modules may be found in Appendix H.

After determining the visibility relations, the next step in object-oriented design is to establish the interface of each object. It was here that the packages began to take shape. I determined the parameters for the operations and coded each package interface.

The final step is to "implement each object." [9] Booch clearly includes this step in design; he is not referring to the implementation phase that is entered upon completion of the design phase. After completing the design for the example system, I can appreciate the significance of including the term "implement" in a design step. Because it is compilable, the Ada design is really the first step of the Ada READY FENCE! implementation. I was very conscious of the fact that I was trying to design, but at the same time, I was writing code. Perhaps this would not be the case for a person having greater experience with Ada. Then the Ada thought patterns would be instinctive and the needs of the design would predominate.

The module designs were shaped by an effort to balance three guidelines:

1. The design must be compilable.

2. The design must express the requirements in formal rather than natural language to eliminate ambiguity.
3. The design must be high-level; coding must be avoided.

This meant constant striving to resolve the confilicting demands of guidelines 2 and 3. The examples in the research were relatively straight forward in order to illustrate the main points involved in the design process. All the requirements could be expressed at a very high level. The Ada design was apparent from the choice of packages and the implementation of the interfaces in the package specifications. At the opposite extreme were the full Ada implementations that Grady Booch supplied for his designs. [10] I was striving for a design somewhere in the middle. I wanted to bring the design just to the level where:

1. The implementor has full details needed to code a module with all required functionality.
2. Those details are expressed formally, with comments limited to aside remarks.

As a result, the design for simple packages such as REPORTS and SCORE_SHEETS is almost a sketch. With the specification of the screens, the structure charts, and the layouts for the reports and score sheets, the implementor needs only the specification of the package interface in order to code. On the other hand, the design for more complex packages such as REGISTRATION contains the definitions of lower level procedures in the package body.

I have included some brief comments describing the functions for these low level procedures. However, the comments serve more as place holders in the design process than as part of the design itself. Not enough detail is included in the comment for the module to be coded; otherwise, the comment would assume the proportions of the natural language design. Here the design really needs to be refined by expansion into the lower level procedure. The design should be continued down to a level that is sufficient to formally express all the requirements. At such a level the functionality could be expressed in a named subroutine call or a terse comment. In extreme cases, perhaps the detail could be expressed formally only by the actual code. Guideline 3 requires that this be avoided where posssible.

The module designs include the package body as well as the interface so that package global data objects may be defined. For top down development, and

to improve readability, the package procedures are defined with the separate clause. Procedures defined with the separate clause are indicated as stubs for which the implementation is deferred. The package specification lists all the procedures compactly. The procedure expansions can follow the package and can be grouped so that a specific procedure is easily located.

READY FENCE!, the top level of the system, was fully designed, defining the system structure. The implementations of CLUBS.CLUB_MENU and CLUBS.ADD_CLUBS are included as examples of the design for lower level modules.

SECTION 10. COMPARISON OF THE NATURAL LANGUAGE AND ADA SPECIFICATIONS

The two designs include the same types of support documents: the user interface specification contained in the screen and report layouts, and structure charts. These are expressions of the requirements but also document the design.

Because I prefer to have similar documentation collected in a single place for reference, I would like to have the Ada design supplemented by some summary of the external data similar to the data dictionary for the natural language design. Reliance upon the screen specifications as documentation of the data is inadequate because the values are not indicated. However, a data dictionary requires separate maintenance, unless it is automatically generated from the design by a support tool. Because of the formality of the Ada design, such a dictionary could be produced. On the other hand, a dictionary is not really necessary for the Ada design, as it is for the natural language design. In Ada the data is more controlled, not global to the system.

A graph of the calling hierarchy is tremendously useful throughout the design phase and again when maintenance is required. The individual structure charts for the Ada packages are well organized illustrations of the calling hierarchy, with procedures grouped by objects in the design. However, the diagrams could become very crowded for complex packages or when many packages interface with the package that is diagrammed. In such a case, the traditional structure chart might be more readable.

10.1. USER APPROVAL AS REAL WORLD MODEL

It is desirable for the client to review the design document and to approve it before implementation begins. At this point design errors and omissions should be revealed. Corrections can then be made to the design rather than to the implementation, which is far more expensive. When the client approves the design, we know that analysis is complete.

On the highest level, the Ada design is more understandable simply because the vocabulary corresponds to real world objects rather than processing steps.

The user can identify the basic design groups and grasp the relationships. This is a definite advantage. Beyond that, the client understands the formal language design not at all; the natural language design communicates only a little better. Like Dante in the Inferno, the client requires a Virgil as guide and interpreter of both designs.

What really interests the client are the low level details that are less easily grasped in the Ada code. What she/he really wants to know are specifics such as the following. Can a fencer with point position be moved in reviewing pools? What choices are offered for pool composition? How are the initial pool assignments determined? Many clients will not be able to answer these questions from either design document. The guide for the Ada design must understand the formal language and explain it to the client. Often the client does not completely understand, and accepts the explanation on faith. The guide for the natural language design must only locate the section and point out the description, which is much more likely to be understood by the client.

The Ada design has the advantage over the natural language design in gaining user approval through execution of the design. The strength of the Ada design is its closeness to code. As a result, a prototype could be produced more rapidly than from the natural language design. The client could then experiment independently with the prototype to answer questions about the system. This direct experience would inspire greater confidence in the product than perusal of a document that merely specifies the product's intended functionality.

10.2. MEETING OBJECTIVES

How do the two designs compare in meeting the objectives of the design specification that were discussed in section 4?

The natural language design is useful as a forward transformation to implementation in any procedural language, including basic Ada. Of course, such an implementation would not take advantage of Ada's real power. The object-oriented Ada design could also be implemented in other languages, but it does not transform well. Without the benefit of encapsulation afforded by the package construct, the Ada design seems wordy and cumbersome. In many languages, the data stores that are encapsulated in each package would become system global, as they are in the natural language design. The Ada design transforms best to an

Ada implementation.

Each design describes the system's algorithms in its own way. The shorter natural language design summarizes, implying certain details without specifying them. The natural language design can say something like "set alpha variables to blanks and numeric variables to zero," but the Ada specification must list each variable name. And instead of saying "for options AL, UC, DC: call corresponding program," the Ada design must have a case statement with an explicit call for each of the three options. The formal language eliminates any ambiguity by forcing the designer to be specific. It also forces the designer to address issues that may be overlooked when an item is implied rather than specified in detail.

Both designs are valid backward references to the system requirements. Each design invents nothing that is not in the requirements. The natural language design fully expresses all the requirements. When the Ada design is completed, it, too, will fully express the requirements. Although the structures of the two systems may differ, the end result of each function must be the same in each design due to the validity of the backward reference. The two designs must be equivalent.

As the documentation component of the software, for a design of this size, it is a matter of preference which design is better documentation for maintaining the system. While the natural language design is readily understandable, the formal language design is more precise. For a much larger design, the formal language would be superior documentation because it would enable the use of tools to provide supplementary information (such as a cross-reference) to improve manageability. I would prefer that the formal language design for each module include a short preface containing design decisions and a brief description of the purpose of the code. This would enhance the understandability of the Ada design and increase its value as documentation.

As documentation for utilizing the system, the natural language design is more useful because it is more readily understood. However, neither design is satisfactory user documentation. The system overview and the specifications for the screens, reports, and forms provide a basis for a user manual. Additional descriptions and directions would be required to supplement these elements of the design specification.

The natural language design is complete in developing all the elements of the requirements specification. The elements of the basic requirements are complete

in the Ada design, but it does not yet express every additional lower level detail. Unless certain areas of the Ada design are developed at lower levels there is not sufficient detail to implement a system that will fully satisfy the client's needs. Developing the remaining elements of the full Ada design would not be difficult and is simply a matter of having time available to do it. Most of the design is already present. The intention was to demonstrate how the Ada design would be expressed and to develop the Ada design structure sufficiently for the purpose of this paper. The natural language design was developed to termination, but not the Ada design. It is not yet ready for implementation.

The Ada design is superior to the natural language design in that its consistency and correctness can be verified. Successful compilation indicates that no conflict exists between parts of the design because it has passed Ada's strict type checking. Execution of the completed Ada design (with stubs for low level modules) could be used to verify that the output relation is true for given input. The natural language design can only be verified by desk checking. Design flaws become fully evident only during the implementation phase.

Traceability is not specific in either design. Neither design contains elements that are not traceable to the requirements document. But the section of the requirements developed in each section of the design is not explicitly cited because I felt that the relation of each design section to a requirements section is obvious. For the design of a larger system, such citations would be required in order to verify the completeness and validity of the design.

Feasibility depends upon the cost of equipment required for implementation so that the system operates within the client's time constraints and the equipment is portable. The natural language design is meant to be implemented in Turbo Pascal. It would be feasible to implement. There is an Ada compiler for micro computers, Augusta. Augusta produces interpreter p-code for a subset of Ada. I do not know exactly what features are provided, but, because Augusta involves interpreted code, it would not be fast enough for READY FENCE! The Ada design is a demonstration of Ada and not a feasible system.

10.3. SYSTEM STRUCTURE

The structures of the two designs differ in organization and size.

The structures of the two systems are roughly similar in respect to the

hierarchy of procedural calls. Modules with similar functions in both designs call corresponding modules with similar functions. In contrast, the essences of the two systems' structures are radically different because of the object orientation of the Ada design. The Ada design looks and feels very distinct from the traditional design. The real structure chart for the Ada design is the visibility chart of Appendix F, not the procedural chart that follows. While the natural language design is a loose collection of modules that might be classified into logical groups, the Ada design is tightly organized into subsystems based on the packages. This organization, which localizes all operations related to an object, enhances the system's understandability and maintainability.

Although I did not complete the entire Ada design to the point where it was ready for implementation, it is evident that the size of the Ada design is greater than the Natural language design. There are 96 modules in the completed natural language design. In the incomplete Ada design there are 100 modules, not counting the packages CONSTANTS and FENCERS, which have no procedures. This 100 includes all the procedures in the package bodies and in procedure READY FENCE!. As the Ada design is developed, there will be additional modules. For EVENTS.REVIEW_POOLS, for example, there will be another 9 sub-programs. These are indicated on page 13 of the Ada READY FENCE! structure chart, but have not been counted because they have not been entered in the design code. For EVENTS.ASSIGN_POOLS there will be another 5 or 6 lower level modules.

Why do we need more modules in Ada to perform identical functions? One answer lies in identifying the objects that perform a function. Because the natural language design has a single set of global data, a single module can perform a function upon multiple items in the global data. In Ada, the data is separated and protected; we need a separate operation to perform the function upon each data object. Instead of having one utility to load tables and one to save tables, we have one to load and one to save each table. In Ada we may have a generic procedure to load a table and another generic procedure to save a table, but we still need separate instantiations; we still need separate modules for each table.

Here is another example of how Ada forces us to split out operations specific to each object into separate procedures. The natural language design specifies that the module for the signon screen accesses the security table to find

the operator's permission and also to change the password if the operator chooses to do so. In Ada, the signon screen module can't access the security table. Instead, the module calls two separate procedures in the OPERATORS package, VALIDATE_OPERATOR and OPERATOR_PASSWORD_CHANGE.

A second answer to the question of the need for more modules in Ada is also associated with Ada's visibility rules, but relates to the object itself rather than operations. We need a module to supply the values for an encapsulated data type. For example, in the natural language design a module can access the global competition record to find out if the competition is open. In Ada, this information is provided by a separate module, the function IS_OPEN in the COMPETITION package. EVENTS.GET_VALUES is another example. It provides procedures in REPORTS with values for individual fields in the event record. In the natural language design, these values are directly available to any module without the aid of an intermediary module.

10.4. LENDING TO IMPLEMENTATION

The Ada design structure seems most appropriate for an Ada implementation; the natural language design structure seems most appropriate for a non-Ada implementation. Within this restriction, both designs have good "code-to ability." Of course, the Ada design would be easier to code because the design itself includes the actual basic code for the upper level modules. The natural language design does not contain actual code, but the pseudo-code is very close to code. Only the lowest levels are not expressed in pseudo code.

The structure of the design must mirror the structure of the implemented system. For this reason, and as a result of my experience with this design, I agree with the Department of Defense. Systems to be implemented in Ada should be designed in Ada in order to take full advantage of the language's features in structuring the system. Conversely, it makes little sense to use Ada's special features, such as packages and tasks, when the system will not be implemented in Ada. The structures used in the design must be limited to those available in the implementation language.

SECTION 11. SUMMARY

11.1. PROBLEMS

Several problems occur in using programming languages, such as Ada, for specification.

11.1.1. COMMENTS CONTAINING INCOMPLETE DESIGN CONCEPTS

Ordinary comments should not contain processing because these might be forgotten when the system is implemented. This problem occurs when pure Ada is used as the design language. Unfinished parts of the design are most conveniently expressed as comments. These are not readily distinguishable from documentary comments included in the implementation. Language extensions are required to eliminate the problem. When formalized comments are used, a tool can be used to identify unfinished parts of the design. Another way to avoid the problem is by using a TBD construct as described in sections 8.4.4 and 8.4.5.

11.1.2. LEVEL OF DETAIL

The most serious problem occurs when a programming language such as Ada is used for design. The degree of detail specified in each language construct is unlimited. The design may be too detailed to be reasonable documentation. [1] There is a need to control the amount of detail in order to keep the design at a high enough level to be descriptive.

In theory, design must not be so detailed as to become code. But when only coding constructs are available, the design must be expressed in code. Restriction of the language to a subset does not eliminate the problem. The more complex the details that must be expressed in the design, the lower the level of the code that is required for that expression. Without extensive commenting, it becomes increasingly difficult to understand the purpose of the code. This defeats the most important purpose of design: to communicate the system.

Geller discusses the conflict caused by the DOD requirement that systems

specifications must be expressed in fully compilable Ada. Using fully compilable Ada for the B-5 functional requirements specification forces the expression either to contain a level of detail that influences the shape of the design, or to contain so little detail as to be meaningless. [2] The government requirement is that the design show only the highest level and not show lower levels of design. Unless the rule is relaxed, the designer is frequently unable to show enough detail to make it clear what the system does. Geller points out the contradiction between the requirement for full Ada and the requirement for high level design only. He argues that the high level requirement should be relaxed to permit more detail.

My conclusion is that an Ada based PDL would fulfill the requirement for compilable code, yet provide constructs to express additional detail. However, the design/coding problem remains. Lindley sums it up:

> "[The] question is that of the distinction between designing in PDL and coding in Ada. Where is the boundary? How is the boundary patrolled? All these questions indicate that while there are obvious benefits to designing programs in a language very similar to a coding language, there are also less obvious pitfalls and difficulties." [3]

11.1.3. COMPATIBILITY OF DESIGN AND IMPLEMENTATION LANGUAGES

When a programming language is used for design, problems arise when the same language is not used for implementation. The design language must be chosen so that it is compatible with the implementation language. It makes little sense to design in a language that does not provide basic structures similar to those included in the implementation language.

The design language must not use constructs that do not map to the implementation language. Designs in Ada should not use Ada specific detailed constructs that may be difficult to simulate using structured code in another language; e.g., design with packages when the implementation will be in FORTRAN. [4]

Conversely, it is possible that the design language may not be able to express a design that can be implemented in another language. IEEE cautions, "An Ada PDL may be used to document a design when the implementation is projected to be in a Programming Language other than Ada. Users should be aware that an Ada PDL may not have the ability to represent such a design." [5]

11.1.4. REUSABILITY

The Ada language has advantages for reusability. [6] It provides improved clarity, reliability, efficiency, and maintainability by embodying and enforcing modern programming practices. Because no subsets or supersets are permitted, it is a stable language; Ada software is portable to any Ada compiler. Ada provides a variety of types of program units. Separation of interface specifications from hidden bodies encourages focus on the module interconnection rather than details of how the module performs its task. Strong typing ensures consistency between formal and actual parameters. The language provides generic program units that can be used as parameterized templates. Program units may be compiled separately and organized in program libraries.

The chief problem in reusing Ada program units is that Ada provides no formal means to specify the unit's functionality.

Litvintchouk and Matsumoto discuss the problem of formally specifying the environment for a reusable Ada component. [7] To reuse an Ada component outside its original system, the user must understand its dependence upon the conceptual structure of its types and functions. Ada permits specification only of interfaces, not semantics. Using pure Ada, informal comments are the only means of communicating the unit's functionality.

Litvintchouk and Matsumoto describe a category theory based language, Clear, that could be used for specifying functionality and environment. A Clear specification defines a "theory" of what a piece of software should do. Clear has operators for building new theories from old ones. It can build a hierarchical "structured" collection of theories. [8] The Clear specification could be automatically mapped to Ada by an APSE tool. [9]

Separate compilation is another obstacle to reusability cited by the same authors. In Ada, if the design is top down, a subprogram body is separately compiled. The definition of the subprogram is elsewhere and must be referenced in the body with a SEPARATE statement. This prevents the component from being multiply reusable. The WITH statement cannot be used because the flow of information is only from the referenced unit to the unit that references it; WITH clauses are not inherited. "Direct application of Ada features leads to either top-down design of systems whose components cannot be reused, or to systems composed of reusable components but which must be constructed in a bottom-up

manner." [10] To avoid the requirement for a particular order of development, Litvintchouk and Matsumoto recommend using instantiations of generic program units with standard visible interfaces. [11]

11.2. OTHER APPROACHES

The thesis project was to express a high level design in pure Ada for comparison with a natural language design for the same system. Other approaches could be taken to assessing Ada as a specification language. It would be interesting to implement an Ada design in another language and then assess the code-to ability of the design. Can the Ada constructs be implemented in structured code? Does the implemented system omit some design constructs that do not lend to structured code?

11.3. CONCLUSIONS

From this study I have drawn some conclusions about using formal languages for specification and about Ada in particular.

11.3.1. COMPLEMENTARY RELATIONSHIP OF NATURAL AND FORMAL LANGUAGE

Natural language and formal language complement each other. They speak to different audiences for different purposes. Natural language is directed toward the end-user and those requiring only a relatively simple overview of the system. Formal language is required for technical development and maintenance of an information system. The formal language specification benefits from a natural language description which explains it. The natural language specification benefits from the insights gained during development of the formal language expression.

11.3.2. ADVANTAGE OF FORMALLY EXPRESSED SPECIFICATIONS

Formal language provides overwhelming advantages in quality and economy. Using a formal language for specification enables early detection and elimination of errors. It permits automatic generation of code, which prevents introduction

of errors in the transition between phases of development. Automatic generation of code reduces software cost and provides flexibility to change specifications. As a result, it becomes economical to customize software, contrary to the current trend of purchasing standardized software because of the high cost of developing customized systems.

11.3.3. LIMITED USE OF NATURAL LANGUAGE

There will be an increasing number of systems that approximate natural language, but these will be for specialized uses such as database query. [12] These systems must overcome the ambiguity and variety of natural language by restricting the set of permissible language or by using an interactive procedure to resolve ambiguity.

Use of formal languages will increasingly replace natural language for specification of software requirements and design. Natural language is ambiguous. Although English is relatively compact, other languages require a larger volume of text than formal languages. [13]

11.3.4. INCREASED USE OF ADA FOR SPECIFICATION

The use of Ada as a program design language will continue to grow. Ada is considered state-of-the-art in language design. [14] The Department of Defense has mandated use of PDL's for design (STD-2167) and requires use of an Ada PDL (DOD directive 3405.2). [15] Ada provides the advantages of a formal language. It strongly supports modern programming practices. It provides the advantages of the Ada environment and support tools. It provides constructs for expressing designs arising naturally from the application. And using Ada for design will aid the continuing transition to using full Ada for system implementation.

APPENDIX A

SYSTEM OVERVIEW

READY FENCE! OVERVIEW

READY FENCE! is an interactive system for managing a fencing competition. A series of menus permit the novice user to perform all operations required to plan and conduct a competition having up to four events. The more experienced user can bypass the menus and transfer directly from screen to screen. The operator may work with multiple events concurrently, transferring between events as needed.

There may be up to 175 fencers registered for each event and up to 25 pools for an event. A fencer may compete in more than one event.

The principal components of the system are:

(1) the security table

(2) the club table

(3) the competition files.

The security table is maintained separately from both the club table and the competition files. It contains the password for each operator and controls the screens that may be accessed by the operator. Access to screens for updating the security table is through an option that is not listed on the system's main menu. The security file is a permanent file; it exists before and after a competition.

The club table contains a record for each club to which fencers are affiliated. The club table may be maintained separately from any competition management. Or a record for a new club may be added to the table when the first fencer from a new club is registered for a competition. The club table is retained and reused for multiple competitions. A menu for maintaining the club table is accessed through the main menu option UC - Update Club Table. A listing of the club data may be printed or displayed on the screen.

The competition files exist only for a particular competition. They contain data that is specific for a particular competition, and are replaced when planning for a new competition begins. There is a one competition file, which contains one record with data pertaining to the general competition, and four records, each of which contains data pertaining to one of the events. For each event, there are also three files: a registration file, a file containing data for rounds of pools, and a file containing data for rounds of direct elimination or elimination with repechage.

Registration files for each event may be opened and built before the day of the competition. Fencers may be added and deleted from the registration table, or their data may be modified, until the assignments are made for the event's first round.

When registration is complete, the operator selects the option to order the

registration list. This assigns a rank in the competition to each fencer according to the fencer's USFA classification and point position, if any.

Then the operator makes assignments for the first round by choosing either the option to assign pools or the option to assign for elimination. For the first round of pools, the system attempts to eliminate placement in the same pool of fencers who have the same club affiliations. The operator would then choose to review the pools for round 1. The review pools option permits the operator to view the pool assignments and to move fencers from one pool to another.

When assignments are complete, the operator would choose to print the forms listing fencers for each pool, or for each strip if the round is one of elimination. These forms are distributed to the scorekeepers to be used for score sheets.

As each pool or strip finishes all their bouts, the completed score sheets are returned. The operator then enters the results through the Input Pools or Input Elimination screen.

When all results are in, the competitors are re-ranked using the Order option. At this point reports can be printed, showing each fencer's current rank in the competition and status of competing or eliminated.

The operator would then choose an option to make assignments for the next round of pools or elimination. In this way, multiple rounds of pools and direct elimination or elimination with repechage may be completed in whatever sequence the operator chooses.

When the last round has been completed, and the last reports produced after ranking, the operator would then close the event. If there were three other events still open, this would permit a new event to be opened.

The operator may withdraw a fencer between rounds.

At any time, the operator can view a display of the status of the competition to see which events are open or closed, or which pools have not yet turned in their score sheets for the current round. An abbreviated display of the competition is displayed each time the operator enters the system after previously logging off. This abbreviated display merely shows the competition title, the titles of the events and their status of open or closed.

A more detailed view of the functions provided by READY FENCE! may be obtained by going through the screens, the data dictionary, and the general descriptions of the system modules.

APPENDIX B

SAMPLE SCREENS

```
(M)                    -)-----   READY FENCE!  MENU  -----(-

     AE    ASSIGN ELIMINATION                              PF1 = MENU
     AP    ASSIGN POOLS                                    PF2 = DEFAULT ACTION
     CR    CORRECT A FENCER'S REGISTRATION DATA           PF10 = EXIT
     DC    DISPLAY COMPETITION STATUS
     DE    DISPLAY ELIMINATION ASSIGNMENTS
     DP    DISPLAY POOL ASSIGNMENTS
     DR    DELETE A REGISTERED FENCER
     IE    INPUT ELIMINATION RESULTS
     IP    INPUT POOL RESULTS
     M     MENU
     O     ORDER LIST OF FENCERS
     PE    PRINT ELIMINATION FORMS
     PP    PRINT POOL FORMS
     PR    PRINT REGISTRATION TABLE
     R     REGISTER FENCERS
     RP    REVIEW POOLS
     TE    TRANSFER TO DIFFERENT EVENT
     UC    UPDATE CLUB TABLE
     W     WITHDRAW A FENCER (BETWEEN ROUNDS)

     E     EXIT FROM READY FENCE!

                                          ENTER NEXT ACTION : __
```

(UC)

UPDATE CLUB TABLE

A	ADD CLUBS TO TABLE
C	CHANGE CLUB
D	DELETE CLUB FROM THE TABLE
L	LIST THE CLUB TABLE ON THE SCREEN (IN NAME ORDER)
P	PRINT THE CLUB TABLE
E	EXIT FROM THE CLUB TABLE UPDATE

ENTER CHOICE OF ACTION: _

ENTER CLUB CODE: ____

NEXT ACTION: __

SECURITY MENU

A ADD OPERATORS

D DELETE OPERATOR

E EXIT SECURITY UPDATE

L LIST ALL OPERATORS

Q OPERATOR INQUIRY

CHOICE: _

(DC) COMPETITION STATUS

COMPETITION: XXXXXXXXX (competition title) XXXXXXXXXX STATUS: OPEN

EVENT 1: XXXXX (event) XXXXXX
EVENT 2: XXXXX (event) XXXXXX
EVENT 3: XXXXX (event) XXXXXX
EVENT 4: XXXXX (event) XXXXXX

EVENT:	1	2	3	4
STATUS - OPEN/CLOSED:	O	C	O	C
RANKED - YES/NO:	Y	N	N	Y
NUMBER REGISTERED:	XXX	XXX	XXX	XXX
NUMBER COMPETING:	XXX	XXX	XXX	XXX
ROUNDS OF POOLS:	XX	XX	XX	XX
NUMBER OF POOLS:	XX	XX	XX	XX
POOLS OUTSTANDING:	XX	XX	XX	XX
ROUNDS OF ELIMINATION:	XX	XX		
ELIMINATION TYPE - DIRECT OR REPECHAGE:	D	R		
ELIMINATION RESULTS OUTSTANDING :	XXX	0		

ENTER Y OR N TO DISPLAY LISTS OF OUTSTANDING POOLS: _
NEXT ACTION: __

(R)

XXXXXXXX (competition title) XXXXXXXXXX

EVENT X: XXXXX (event) XXXXXX

FENCER REGISTRATION

ID: NNN
NAME: _____
MAJOR CLUB CODE: ____
SECOND CLUB CODE: ____
UNATTACHED? ENTER Y OR N: _
USFA RANK: _
POINT POSITION: __

CORRECT? ENTER Y OR N: _

NEXT ACTION: __

(W)

XXXXXXXX (competition title) XXXXXXXXXX
EVENT X: XXXXX (event) XXXXXX

WITHDRAW FENCER

ID:	XXX
NAME:	XXXXXXXXXXXXXXXXXXXX
MAJOR CLUB:	XXXX
SECOND CLUB:	XXXX
UNATTACHED:	X
USFA RANK:	X
POINT POSITION:	X
BOUTS:	XX
VICTORIES:	XX
VICTORIES/BOUTS:	.XXX
HITS SCORED:	XX
HITS RECEIVED:	XX
INDICATOR:	SXX
RANK:	XXX

ENTER Y OR N TO WITHDRAW: _
FENCER XXX HAS BEEN WITHDRAWN

NEXT ACTION: __

(AP) ASSIGN POOLS

NUMBER COMPETING IS NNN

SOME POSSIBLE CHOICES FOR NUMBER OF POOLS:

```
FOR POOLS OF 4, THERE WOULD BE XX POOLS OF 4 AND XX POOLS OF 5  TOTAL = XX
FOR POOLS OF 5, THERE WOULD BE XX POOLS OF 5 AND XX POOLS OF 6  TOTAL = XX
FOR POOLS OF 5, THERE WOULD BE XX POOLS OF 5 AND XX POOLS OF 4  TOTAL = XX
FOR POOLS OF 6, THERE WOULD BE XX POOLS OF 6 AND XX POOLS OF 7  TOTAL = XX
FOR POOLS OF 6, THERE WOULD BE XX POOLS OF 6 AND XX POOLS OF 5  TOTAL = XX
FOR POOLS OF 7, THERE WOULD BE XX POOLS OF 7 AND XX POOLS OF 6  TOTAL = XX
```

ENTER TOTAL NUMBER OF POOLS: __

THERE WILL BE NN POOLS OF N AND NN POOLS OF N FENCERS

ASSIGNING POOLS . . .
POOLS ASSIGNED

ENTER NEXT ACTION: __

```
(RP) EVENT X                          REVIEW POOLS                    NUMBER OF POOLS: XX
                    POOL  1                                                POOL  2
1 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 1 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
2 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 2 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
3 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 3 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
4 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 4 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
5 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 5 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
6 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 6 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
                    POOL  3                                                POOL  4
1 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 1 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
2 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 2 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
3 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 3 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
4 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 4 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
5 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 5 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
6 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 6 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
                    POOL  5                                                POOL  6
1 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 1 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
2 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 2 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
3 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 3 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
4 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 4 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
5 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 5 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
6 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX 6 XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
            FENCER  ON HOLD:  XXXXXXXXXXXXXXXXXXXX X* XXX XXXX XXXX
USE PF KEYS: 1=APPROVE  3=HOLD 4=INSERT 5=MOVE 6=SWAP  7=FORWARD 8=BACK 9=GOTO
```

(DE)

ELIMINATION ASSIGNMENTS ROUND X
EVENT X: XXXXX (event) XXXXXX

#	FENCER	CLUB	RANK	#	FENCER	CLUB	RANK
1	XXXXXXXXXXXXXXXXXXXX	XXXX	XX	9	XXXXXXXXXXXXXXXXXXXX	XXXX	XX
2	XXXXXXXXXXXXXXXXXXXX	XXXX	XX	10	XXXXXXXXXXXXXXXXXXXX	XXXX	XX
3	XXXXXXXXXXXXXXXXXXXX	XXXX	XX	11	XXXXXXXXXXXXXXXXXXXX	XXXX	XX
4	XXXXXXXXXXXXXXXXXXXX	XXXX	XX	12	XXXXXXXXXXXXXXXXXXXX	XXXX	XX
5	XXXXXXXXXXXXXXXXXXXX	XXXX	XX	13	XXXXXXXXXXXXXXXXXXXX	XXXX	XX
6	XXXXXXXXXXXXXXXXXXXX	XXXX	XX	14	XXXXXXXXXXXXXXXXXXXX	XXXX	XX
7	XXXXXXXXXXXXXXXXXXXX	XXXX	XX	15	XXXXXXXXXXXXXXXXXXXX	XXXX	XX
8	XXXXXXXXXXXXXXXXXXXX	XXXX	XX	16	XXXXXXXXXXXXXXXXXXXX	XXXX	XX

NEXT ACTION: __

(IP)

XXXXXXXX (competition title) XXXXXXXXXX
EVENT X: XXXXX (event) XXXXXX

INPUT POOL XX

ROW X CLUB XXXX FENCER XXXXXXXXXXXXXXXXXXXX

VICTORIES: _

BOUTS: _

HITS SCORED: __

HITS RECEIVED: __

QUALIFIED/ELIMINATED (Q OR E): _

IS DATA CORRECT? _

NEXT ACTION: __

APPENDIX C

REQUIREMENTS DEFINITION

READY FENCE! REQUIREMENTS

CONTEXT

Problem

A fencing event is marked by lengthy intervals between rounds while results are compiled, competitors are ranked and eliminated or promoted, and assignments are made for the next round. Ranking is especially time consuming. The larger the number of competitors, the longer it takes to prepare for the next round. The fencing bouts themselves consume many hours. Delays between rounds can lengthen an event beyond the limit of acceptability.

Needs

The process of preparation for the next round needs to be accelerated. This would shorten the length of a competitive event and would permit larger numbers of competitors without unacceptably increasing the duration of the event.

Objectives

Develop a system that would automate the ranking and assignment functions without increasing the time required for compiling results of the previous round. The system must produce forms and reports in no more time than is required for manual preparation.

FUNCTIONAL SPECIFICATIONS

1. Maintain the status of a competition consisting of four independent concurrent events.

2. Permit the operator to control the format of the event by choosing rounds of pools and/or elimination in any sequence. Elimination rounds may be direct elimination or elimination with repechage.

3. Register up to 175 fencers for each event. A fencer may be affiliated with two clubs and also may choose to fence without attachment to a club with which she/he is affiliated. USFA classifications are A, B, C, D, E, or U (unclassified). The top A fencers will have a point position which is a relative USFA ranking based on past competitions. Each registered fencer has an initial status of C, competing.

4. Cancel a fencer's registration if requested before competition opens. The registration record for a cancelled fencer should be deleted.

5. Withdraw a fencer from an event if requested after competition opens. Status for a withdrawn fencer is changed to Q. Q indicates that the fencer should be included in the next ranking and then should have status changed to W, withdrawn.

6. Rank fencers as a basis for bout assignments. Update the rank number for each fencer.

   6.1 Before competition begins, rank all fencers by USFA classification and point position. Highest rank is class A fencer with lowest point position (A1). Unclassified fencers have lowest rank.

   6.2 After completion of the first round, rank by results of previous matches. Highest rank is fencer with highest Victories/Bouts. In case of tie, highest rank is fencer with highest (Hits Scored - Hits Received). In case of tie, highest rank is fencer with lowest Hits Received. All fencers who competed in the previous round must be ranked. These have status C, K, and Q. After sorting fencers with status K should be changed to E (eliminated) and those with Q should be changed to W (withdrawn).

   6.3 In preparation for round 3 of elimination with repechage, only losers in the winner list and winners in the loser list are ranked. These have status R. After sorting, fencers with status R should be changed to C (competing).

7. Assign fencers to pools of 4 - 7 on the basis of rank according to F.I.E. rules. Assignment of competitors marks the beginning of a new round of pools.

   7.1 For the first round of pools, present the operator with six possible combinations of number of fencers per pool. The operator may choose one of these or specify another. The algorithm for determining the combinations is described in a separate section at the end of the requirements.

   7.2 For the first round of pools distribute fencers in the pools by rank from the highest (rank 1) to the lowest (rank N). Place one fencer in each pool until there are no more pools. Then place the next fencer in the same pool as the last fencer and continue placing fencers in the pools in the reverse order until there are no more pools. Continue placing fencers and reversing direction of placement as necessary until all fencers have been placed.

   For the first round of pools only, attempt to eliminate multiple fencers from the same affinity group in the same pool by swapping the second fencer with a fencer in another pool. Class A fencers with point position may not be moved from the initial pool placement by rank. Other fencers may be swapped with another fencer of the same USFA class. If the multiple can not be eliminated by a single swap, attempt a double swap. If a double swap would not eliminate the multiple, then leave the pool as initially composed. A description of the swapping process is included in a separate section at the end of the requirements.

8. For the first round of pools only, after initial assignment of fencers to pools, permit the operator to review pool assignments and to rearrange the

placements, if desired. Class A fencers with point position may not be moved from the initial pool placement by rank.

9. Assign fencers to bouts of direct elimination or elimination with repechage according to F.I.E. rules. Assignment of competitors marks the beginning of a new round of elimination.

   For elimination, the number of competitors must be 32, 16, or 8. For direct elimination, distribute competing fencers according to rank in the list of elimination bouts from the top to the bottom of the list as follows.

   8 competitors: Order of ranks is
   1,8,5,4,3,6,7,2.

   16: Order of ranks is
   1,16,9,8,5,12,13,4,
   3,14,11,6,7,1,10,15,2.

   32: Order of ranks is
   1,32,17,16,9,24,25,8
   5,28,21,12,13,20,29,4,
   3,30,19,14,11,22,27,6,
   7,26,23,10,15,18,31,2.

   A different method of assignment is used for round 2 of elimination with repechage. Elimination with repechage is described in a separate section at the end of the requirements.

10. Print score sheets for rounds of pools and elimination. Forms contain the names and numbers of assigned fencers. For pools, club affiliation is also included.

    For rounds of elimination, take pairs of fencers in order from the elimination list. If score sheets are for round 2 of elimination with repechage, take pairs of fencers from the winner and loser lists. See the description of repechage at the end of the requirements.

    Score sheets for a round of elimination for eight fencers should include spaces for the last two rounds. The scorekeepers will write in the names of the fencers for these two rounds.

11. Update fencers' registration data with results of bouts for each round. The operator may choose to include or not to include results of previous rounds in cumulative data. Data to be updated is number of bouts, number of victories, the proportion of victories for bouts, number of hits scored, number of hits received, and the difference between hits scored and hits received. The operator indicates whether a fencer should be eliminated from the competition as a result of the previous round.

    Fencers to be eliminated should have their status changed to K. This indicates that they should be included in the next ranking and then have their status changed to E, eliminated.

When input is for a round of direct elimination, remove each loser from the elimination list and move the following fencers upward in the list so that there are no blank places in the list.

When input is for a round of elimination that includes only eight fencers, prompt for input of data for the final two rounds of elimination after input of data for the round of eight fencers.

12. Print a report of all fencers' status for an event in order by rank and a report in order by name.

13. Maintain a list of clubs with which fencers are affiliated. Organize clubs into affinity groups of associated clubs. A club's affinity group may be itself.

14. Print a report of all clubs with their affinity group. Print in order by club name and in order by club code.

15. Maintain a list of operators authorized to use the system. For each operator record a password and one of three levels of access: read only, update, and security update.

16. Limit read, update, and security update actions to authorized operators only.

MANUAL FUNCTIONS

1. Score sheets are filled in by the scorekeeper for each strip.

2. Completed score sheets are returned to the bout committee for data entry as soon as the bouts for the strip are completed.

3. The bout committee determines whether a competitor is to be eliminated at the end of a round or promoted to the next round.

4. After completion of a round of elimination with 8 fencers, the scorekeeper writes in the names of the competitors participating in the next two rounds. After completion of the final round, data for each of the three rounds is input successively.

CONSTRAINTS

Performance

1. Execution speed must be such that the time to prepare the next round of an event is less than the time required when done manually.

2. Printer speed must be sufficient to print a pool form in 40 seconds or less.

Reliability

1. Data must not be lost if the system goes down due to power loss.

Security

1. Unauthorized persons must not be able to enter the system to tamper with data.

ALGORITHM FOR DETERMINING 6 POOL CHOICES

1. pools of 4 - method 1: put remainder in other pools

```
divide number competing by 4
     giving quotient + remainder

/* there will be A pools of 4 and B pools of 5

A = quotient - remainder
B = remainder

/*  e.g.  83/4 = 20 remainder 3

          17 pools of 4 = 68
           3 pools of 5 = 15
          20              83

display
     "FOR POOLS OF 4, THERE WOULD BE " A " POOLS OF 4"

IF B is greater than 0
     display " AND " B " POOLS OF 5"

total pools = A + B

display "TOTAL POOLS: " total pools
```

2. pools of 5 - method 1

```
divide number competing by 5
     giving quotient + remainder

/* there will be A pools of 5 and B pools of 6

A = quotient - remainder
B = remainder

/*  e.g.  83/5 = 16 remainder 3
          13 pools of 5 = 65
           3 pools of 6 = 18
          16              83

display
     "FOR POOLS OF 5, THERE WOULD BE " A " POOLS OF 5"

IF B is greater than 0
     display " AND " B " POOLS OF 6"

total pools = A + B
display "TOTAL POOLS: " total pools
```

```
3.   pools of 5 - method 2:
          take from other pools and put into remainder pool

     /* there will be A pools of 5 and B pools of 4

     A = quotient - B + 1
     B = 5 - remainder

     /*  e.g.  83/5 = 16 remainder 3
               15 pools of 5 = 75
               _2 pools of 4 = _8
               17              83

     display
          "FOR POOLS OF 5, THERE WOULD BE " A " POOLS OF 5"

     IF B is greater than 0
          display " AND " B " POOLS OF 4"

     total pools = A + B
     display "TOTAL POOLS: " total pools

4.   pools of 6 - method 1

     divide number competing by 6
          giving quotient + remainder

     /* there will be A pools of 6 and B pools of 7

     A = quotient - remainder
     B = remainder

     /*  e.g.  83/6 = 13 remainder 5
               _8 pools of 6 = 48
               _5 pools of 7 = 35
               13              83

     display
          "FOR POOLS OF 6, THERE WOULD BE " A " POOLS OF 6"

     IF B is greater than 0
          display " AND " B " POOLS OF 7"

     total pools = A + B
     display "TOTAL POOLS: " total pools
```

5. pools of 6 - method 2

```
divide number competing by 6
     giving quotient + remainder

/* there will be A pools of 6 and B pools of 5

A = quotient - B + 1
B = 5 - remainder

/*  e.g.  83/6 = 13 remainder 3
          13 pools of 6 = 78
          _1 pools of 5 = _5
          14              83

display
     "FOR POOLS OF 6, THERE WOULD BE " A " POOLS OF 6"

IF B is greater than 0
     display " AND " B " POOLS OF 5"

total pools = A + B
display "TOTAL POOLS: " total pools
```

6. pools of 7 - method 2

```
divide number competing by 6
     giving quotient + remainder

/* there will be A pools of 7 and B pools of 6

A = quotient - B + 1
B = 5 - remainder

/*  e.g.  83/7 = 11 remainder 6
          11 pools of 7 = 77
          _1 pools of 6 = _6
          14              83

display
     "FOR POOLS OF 7, THERE WOULD BE " A " POOLS OF 7"

IF B is greater than 0
     display " AND " B " POOLS OF 6"

total pools = A + B
display "TOTAL POOLS: " total pools
```

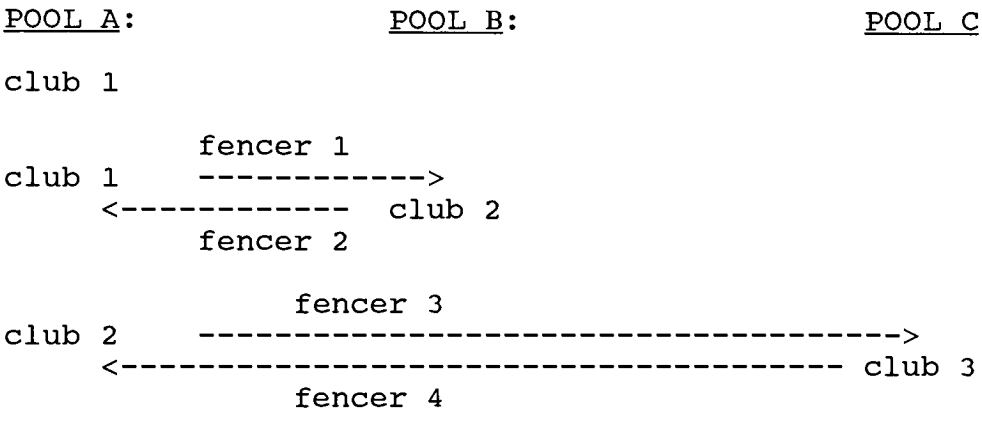
SWAPPING FENCERS TO ELIMINATE DUPLICATE CLUBS

A duplicate club means that two fencers in the same pool have a common value for the affinity group for the major club or the second club.

All fencers involved in a swap have the same USFA classification. The class composition of each pool may not be altered.

A single swap means to swap one fencer in pool A with one fencer in pool B.

A double swap means to swap fencer 1 in pool A with fencer 2 in pool B, eliminating duplicate club 1 (fencer 1's club). This causes duplicate club 2 in pool 1 (fencer 2's club). But we can swap the duplicate, fencer 3 in pool 1, with fencer 4 in pool C without causing another duplicate club in pool 1.



For the swap to be feasible, the following must all be true:

- There is no club 1 in pool B.
- There is only one club 2 in pool 1.
- There is no club 2 in pool 3.
- There is no club 3 in pool 1.

We don't try to put fencer 3 into pool 2 because if there is a fencer in pool 2 with the same USFA class and a club that is not in pool 1, that fencer would have been found in the previous attempt at a single swap.

Double swap is a last resort when all attempts fail for a single swap in a given pool.

ACTIVITIES FOR ELIMINATION WITH REPECHAGE

For Elimination with repechage, as for direct elimination, the number of competitors must equal 8, 16, or 32 only.

RANK

The registration list must be ranked before assignments for round 1.

ROUND 1

ASSIGN

Assignment indicates the beginning of round 1. Elimination assignments for round 1 are the same as for direct elimination. An elimination list is built according to rank. All competitors are assigned by appropriate placement in the elimination list.

PRINT ELIMINATION FORMS

Print elimination forms for round 1 using the elimination list.

INPUT RESULTS

Input results of round 1. Update the competitors registration records, and insert fencers into the winner and loser lists.

ROUND 2

ASSIGN

Assignment indicates the beginning of round 2. No assignment is needed for this round. All competitors participate in this round.

PRINT ELIMINATION FORMS

Print elimination forms for round 2 using the winner and loser lists. Take pairs of fencers in order from the winner list first; then take pairs of fencers from the loser list.

INPUT RESULTS

Input results of round 2. Update the competitors registration records. Update the status for each competitor as follows.

status = C (no change) for winner in winner list
= K for loser in loser list
= R for winner in loser list
= R for loser in winner list

Count the number with R status. This is the number of competitors in the repechage round which follows.

RANK

Round of elimination is still 2. Only rank fencers with status R; fencers with status C and K are not in the rank list - they will be ranked later. Fencers with status R will participate in the repechage round. After ranking, status R is changed back to C.

ROUND 3 (Repechage Round)

ASSIGN

Assignment indicates the beginning of round 3. Elimination assignments for round 3 are the same as for round 1 except that the number of fencers included is the number counted after round 2 (those with status R, all of whom were included in the ranking). A new elimination list is built according to rank. All fencers designated with status R at the close of round 2 are assigned to bouts by appropriate placement in the elimination list.

PRINT ELIMINATION FORMS

Print elimination forms for round 3 using the elimination list.

INPUT RESULTS

Input results of round 3. Update the competitors registration records. Update the status for each competitor as follows.

status = K for losers (rank, then eliminate)
= C for winners (competing)

RANK

Round of elimination is still 3. Fencers with status C and K are ranked. Fencers with status K are ranked and then set to E, eliminated.

ROUND 4

ASSIGN

Assignment indicates the beginning of round 4. Elimination assignments for round 4 are the same as for round 1. A new elimination list is built according to rank using all surviving competitors. These include the winners of the winner list from round 2 and the winners of the repechage round in round 3. Each fencer has had two wins. All competitors are assigned by appropriate placement in the elimination list.

PRINT ELIMINATION FORMS

Print elimination forms for round 4 using the elimination list.

INPUT RESULTS

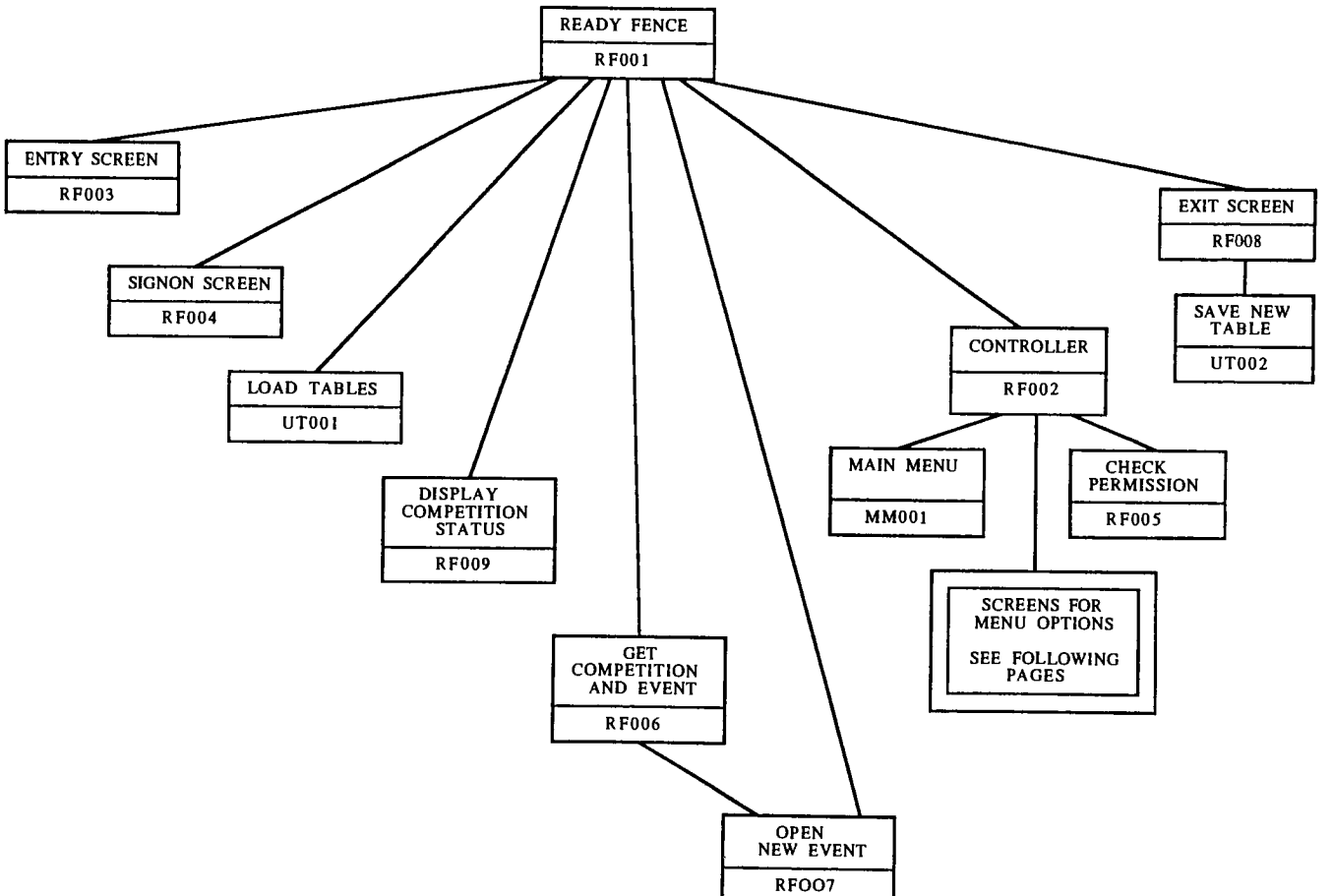
Input results of round 4 and results of final bouts. Update the competitors registration records.

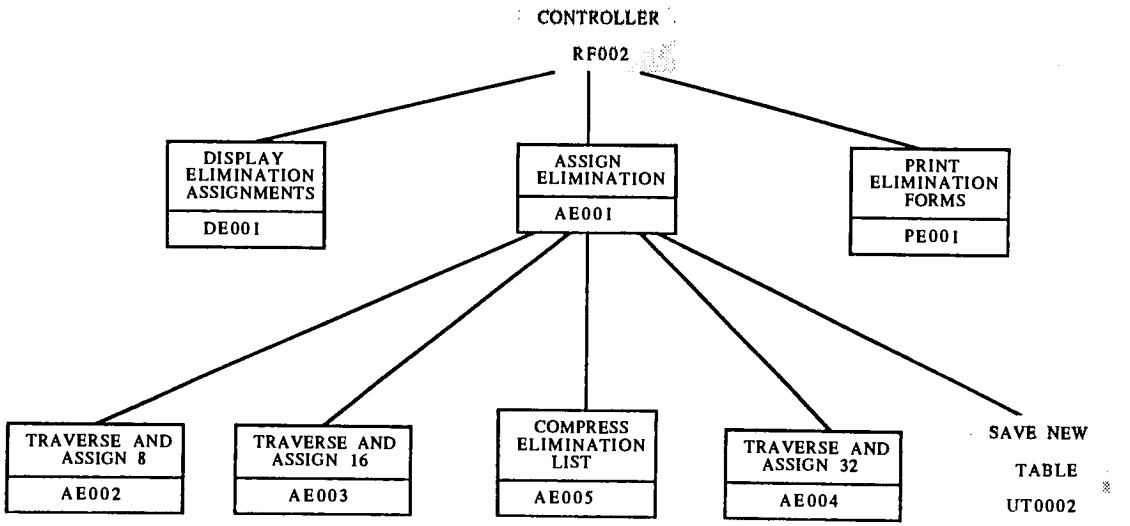
APPENDIX D

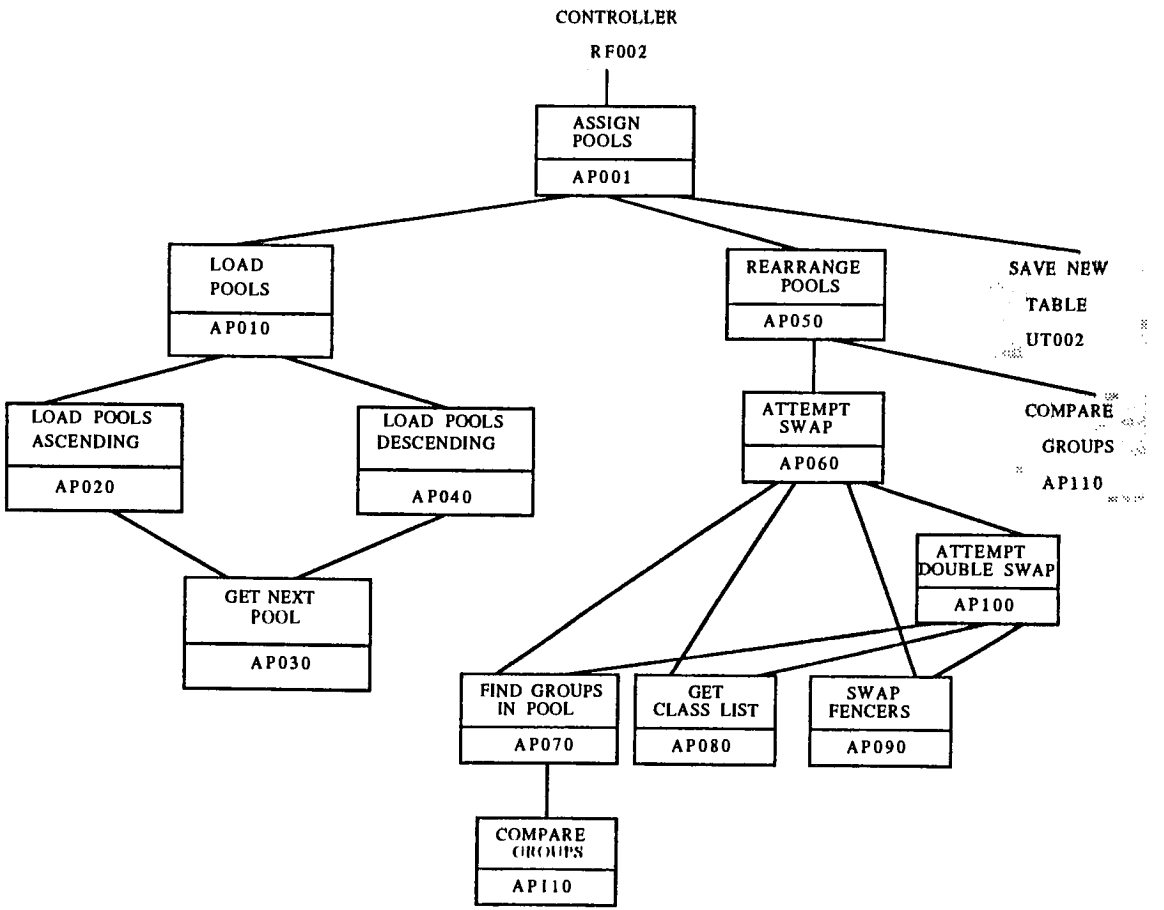
STRUCTURE GRAPHS FOR NATURAL LANGUAGE DESIGN

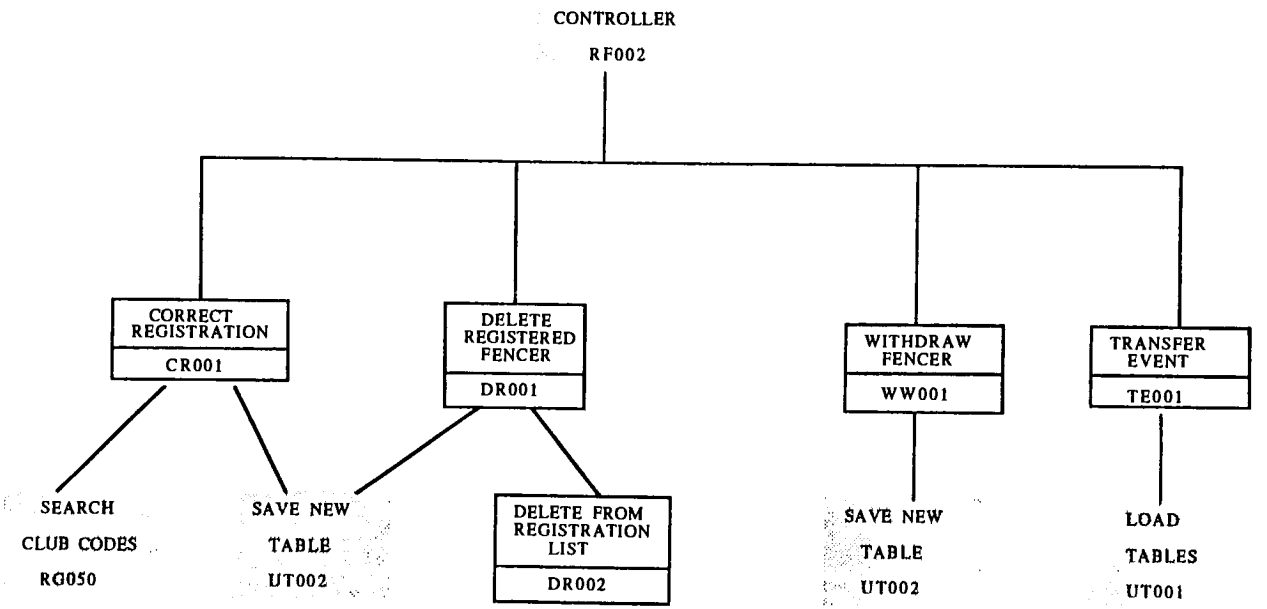
READY FENCE! STRUCTURE

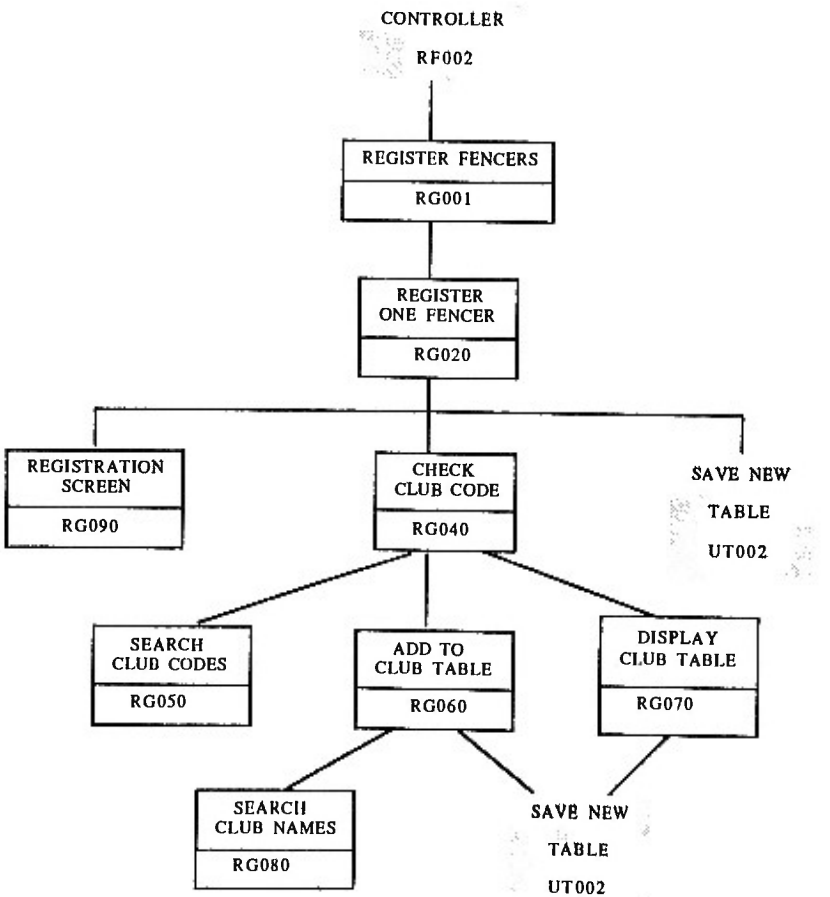
INDEX

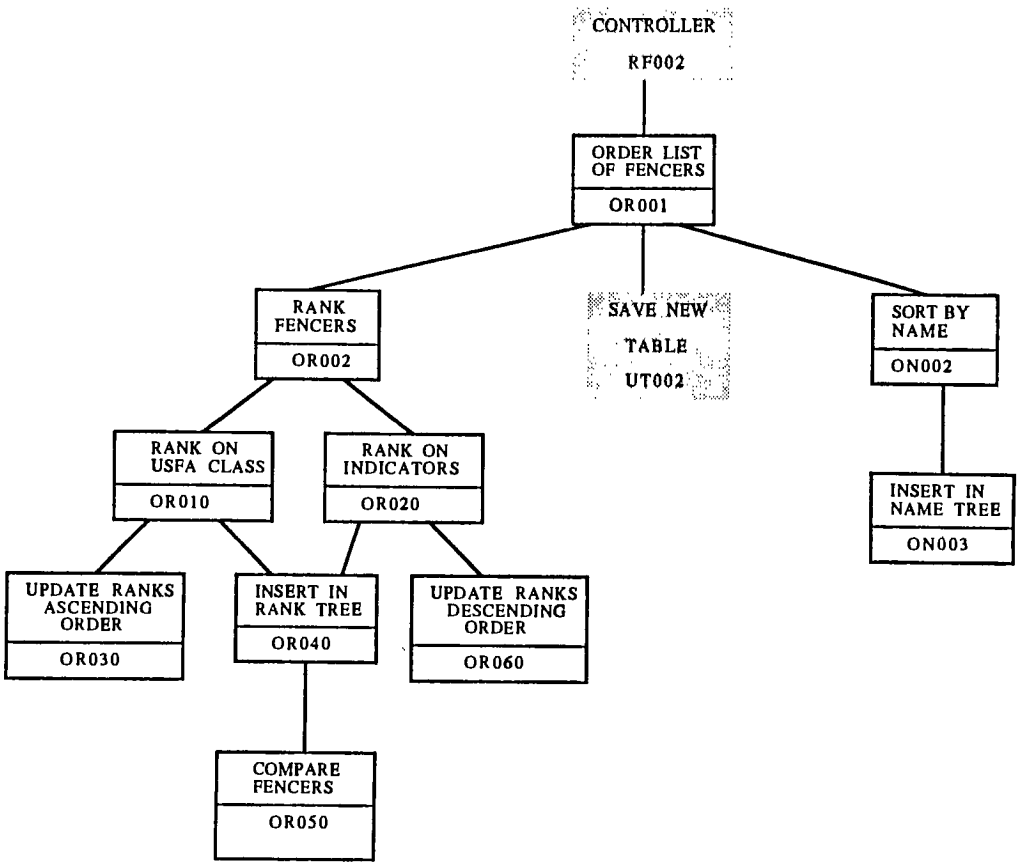

READY FENCE
RF001
ENTRY SCREEN
RF003
SIGNON SCREEN
RF004
LOAD TABLES
UT001
DISPLAY COMPETITION STATUS
RF009
GET COMPETITION AND EVENT
RF006
OPEN NEW EVENT
RFOO7
CONTROLLER
RF002
MAIN MENU
MM001
CHECK PERMISSION
RF005
SCREENS FOR MENU OPTIONS
SEE FOLLOWING PAGES
EXIT SCREEN
RF008
SAVE NEW TABLE
UT002

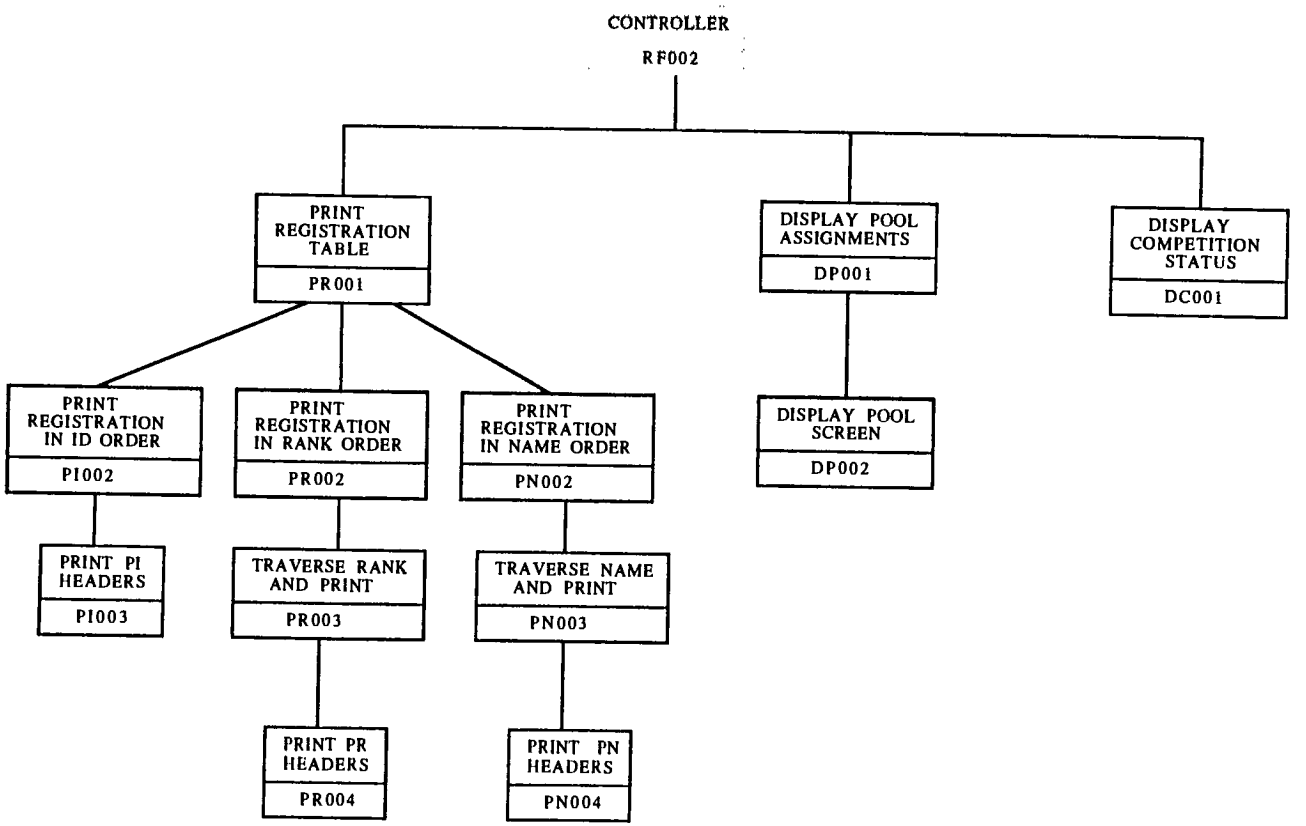

CONTROLLER
RF002
DISPLAY ELIMINATION ASSIGNMENTS
DE001
ASSIGN ELIMINATION
AE001
PRINT ELIMINATION FORMS
PE001
TRAVERSE AND ASSIGN 8
AE002
TRAVERSE AND ASSIGN 16
AE003
COMPRESS ELIMINATION LIST
AE005
TRAVERSE AND ASSIGN 32
AE004
SAVE NEW TABLE
UT0002

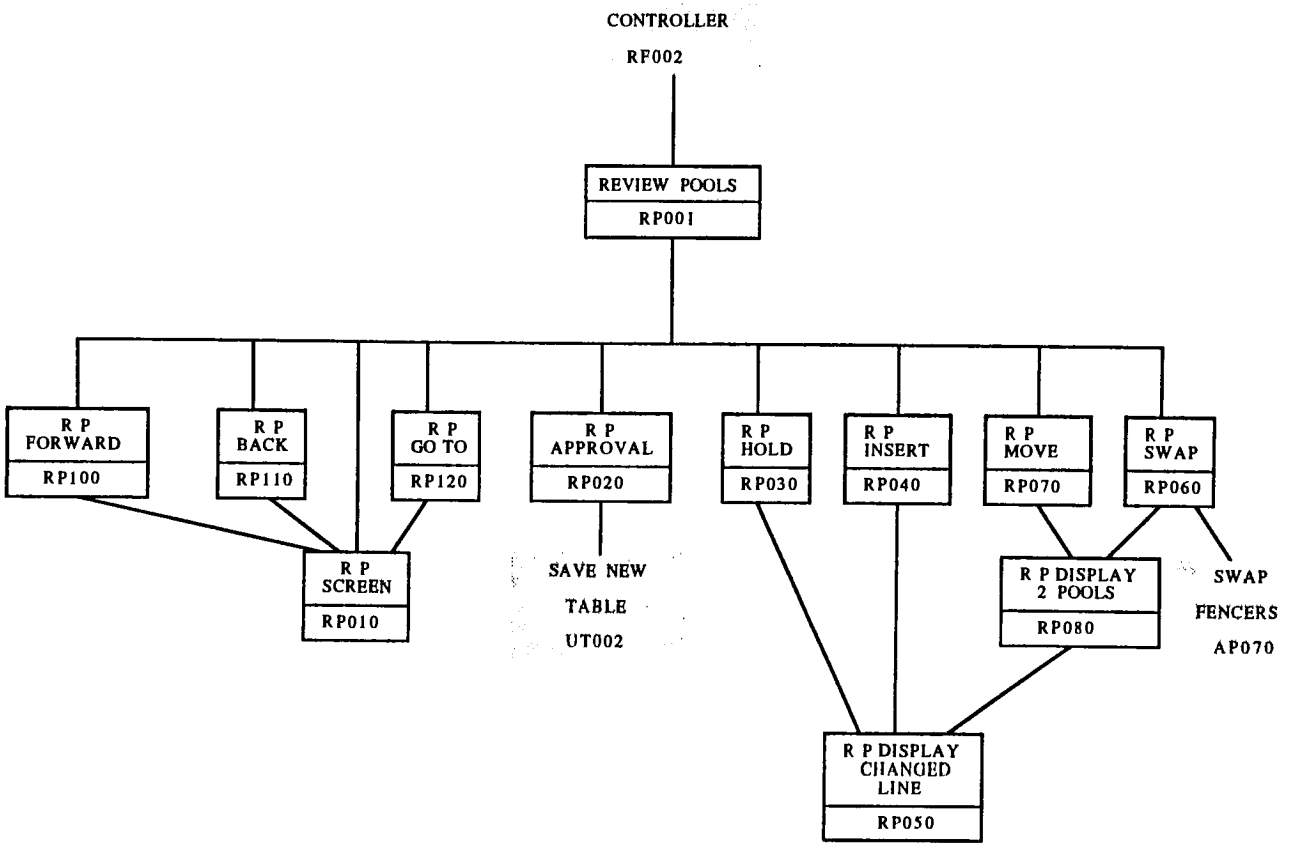

CONTROLLER
RF002
ASSIGN POOLS
AP001
LOAD POOLS
AP010
REARRANGE POOLS
AP050
SAVE NEW TABLE
UT002
LOAD POOLS ASCENDING
AP020
LOAD POOLS DESCENDING
AP040
ATTEMPT SWAP
AP060
COMPARE GROUPS
AP110
ATTEMPT DOUBLE SWAP
AP100
GET NEXT POOL
AP030
FIND GROUPS IN POOL
AP070
GET CLASS LIST
AP080
SWAP FENCERS
AP090
COMPARE GROUPS
AP110

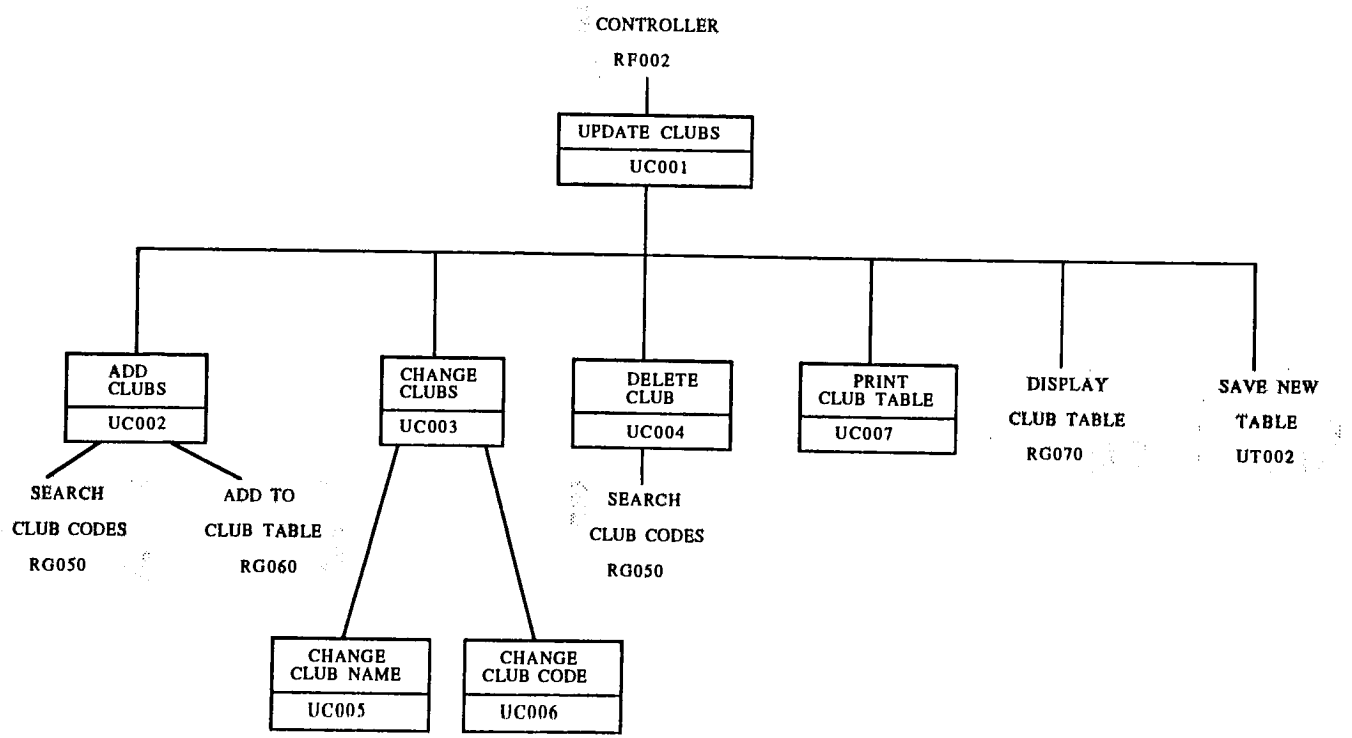

CONTROLLER
RF002
CORRECT REGISTRATION
CR001
DELETE REGISTERED FENCER
DR001
WITHDRAW FENCER
WW001
TRANSFER EVENT
TE001
SEARCH CLUB CODES
RG050
SAVE NEW TABLE
UT002
DELETE FROM REGISTRATION LIST
DR002
SAVE NEW TABLE
UT002
LOAD TABLES
UT001

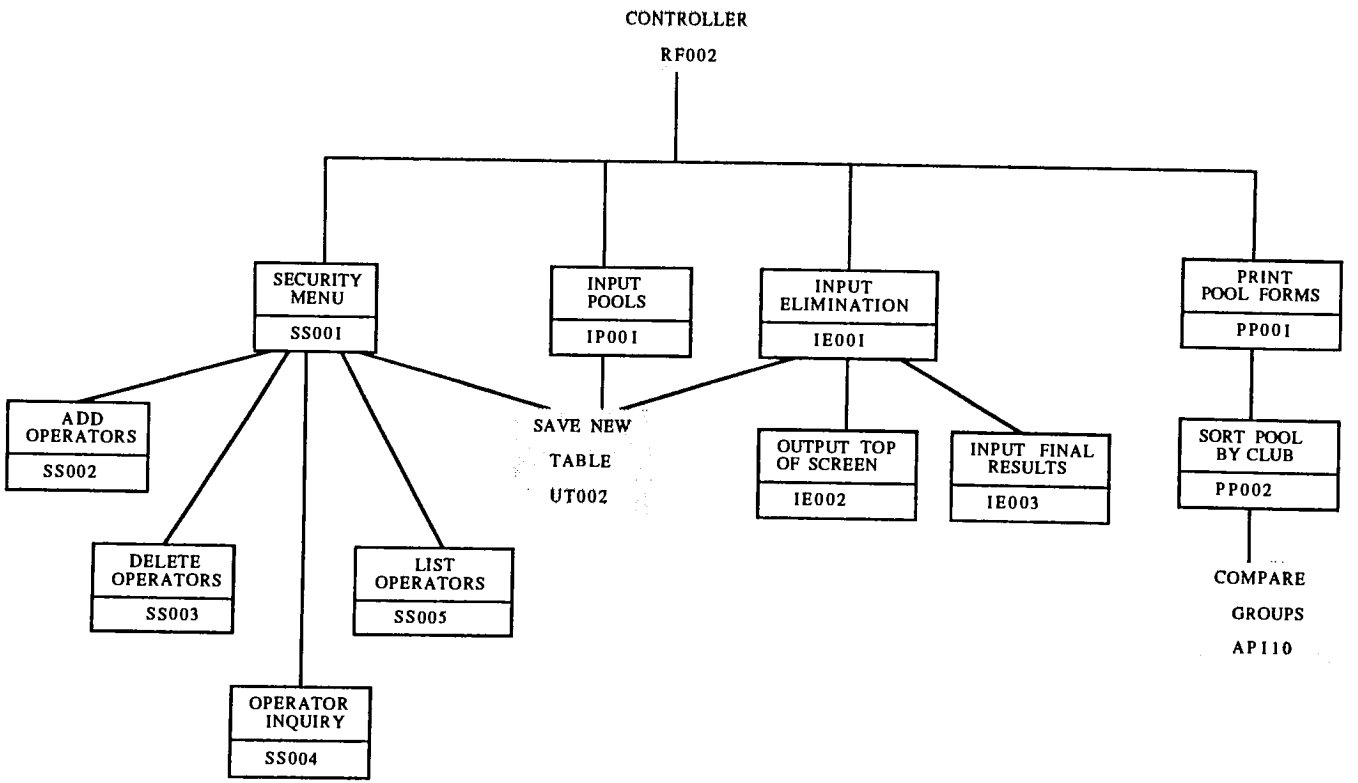

CONTROLLER
RF002
REGISTER FENCERS
RG001
REGISTER ONE FENCER
RG020
REGISTRATION SCREEN
RG090
CHECK CLUB CODE
RG040
SAVE NEW TABLE
UT002
SEARCH CLUB CODES
RG050
ADD TO CLUB TABLE
RG060
DISPLAY CLUB TABLE
RG070
SEARCH CLUB NAMES
RG080
SAVE NEW TABLE
UT002


CONTROLLER
RF002
ORDER LIST OF FENCERS
OR001
RANK FENCERS
OR002
SAVE NEW TABLE
UT002
SORT BY NAME
ON002
RANK ON USFA CLASS
OR010
RANK ON INDICATORS
OR020
INSERT IN NAME TREE
ON003
UPDATE RANKS ASCENDING ORDER
OR030
INSERT IN RANK TREE
OR040
UPDATE RANKS DESCENDING ORDER
OR060
COMPARE FENCERS
OR050


CONTROLLER
RF002
PRINT REGISTRATION TABLE
PR001
DISPLAY POOL ASSIGNMENTS
DP001
DISPLAY COMPETITION STATUS
DC001
PRINT REGISTRATION IN ID ORDER
PI002
PRINT REGISTRATION IN RANK ORDER
PR002
PRINT REGISTRATION IN NAME ORDER
PN002
DISPLAY POOL SCREEN
DP002
PRINT PI HEADERS
PI003
TRAVERSE RANK AND PRINT
PR003
TRAVERSE NAME AND PRINT
PN003
PRINT PR HEADERS
PR004
PRINT PN HEADERS
PN004


CONTROLLER
RF002
REVIEW POOLS
RP001
R P FORWARD
RP100
R P BACK
RP110
R P GO TO
RP120
R P APPROVAL
RP020
R P HOLD
RP030
R P INSERT
RP040
R P MOVE
RP070
R P SWAP
RP060
R P SCREEN
RP010
SAVE NEW TABLE
UT002
R P DISPLAY 2 POOLS
RP080
SWAP FENCERS
AP070
R P DISPLAY CHANGED LINE
RP050


CONTROLLER
RF002
UPDATE CLUBS
UC001
ADD CLUBS
UC002
CHANGE CLUBS
UC003
DELETE CLUB
UC004
PRINT CLUB TABLE
UC007
DISPLAY CLUB TABLE
RG070
SAVE NEW TABLE
UT002
SEARCH CLUB CODES
RG050
ADD TO CLUB TABLE
RG060
SEARCH CLUB CODES
RG050
CHANGE CLUB NAME
UC005
CHANGE CLUB CODE
UC006


CONTROLLER
RF002
SECURITY MENU
SS001
INPUT POOLS
IP001
INPUT ELIMINATION
IE001
PRINT POOL FORMS
PP001
ADD OPERATORS
SS002
SAVE NEW TABLE
UT002
OUTPUT TOP OF SCREEN
IE002
INPUT FINAL RESULTS
IE003
SORT POOL BY CLUB
PP002
DELETE OPERATORS
SS003
LIST OPERATORS
SS005
COMPARE GROUPS
AP110
OPERATOR INQUIRY
SS004

APPENDIX E

NATURAL LANGUAGE DESIGN FOR HIGH LEVEL MODULES

RF001 Entry and Exit

Purpose:

Provide an entry point to the system.

Control activities at entry and prior to menu.

Provide an exit point that will permit the user to exit from an event but resume with a different event, if the current event is closed.

Signon screen module loads the security file and gets the user's security level (permission). This module, RF001, loads the competition control file. Module to get competition and event loads the tables specific for the event.

The user may wish to update tables at a time when no competition is being managed. At entry, the user is asked to establish an action class of C or T. T (table actions only) permits updating tables when a competition is not being managed. Only table actions are permitted. C (competition actions) permits any action, including table actions.

Detailed Specification:

```
display entry screen                                  /* SUBROUTINE
display signon screen (accept, permission)            /* SUBROUTINE
IF accept = "N"
     terminate

/* load competition file
load tables (A)                                       /* SUBROUTINE
N clubs = 0
restart = " "

REPEAT until restart = "N"
     IF competition is open
          action class = "C"
          display competition status                  /* SUBROUTINE
          get competition and event                   /* SUBROUTINE
     ELSE
          display "ENTER C TO OPEN COMPETITION OR T TO
UPDATE TABLES"
          input action class
          IF action class = "C"
               output "ENTER TITLE OF COMPETITION"
               input competition title
               event = 1
               output "ENTER TITLE OF EVENT"
               input event title (1)
               open new event                         /* SUBROUTINE
     IF N clubs = 0           /* club table is not loaded
          IF (N rounds pools (event) = 0   /* registration
             and N rounds elimination (event) = 0   /* is
                                                    /* open
             and rank indicator (event) = "N")
             OR
             (reply = "T")
                    /* load club file
                    load tables (C)                   /* SUBROUTINE
     next action = "M"    /* menu
     call controller action class, permission) /* SUBROUTINE
     send exit screen (restart)                       /* SUBROUTINE
END of repeat loop    /* user selects exit,
                      /* i.e. restart = "N"
terminate
```

RF002 Controller (action class, permission)

Purpose:

Call modules for menu actions until user selects exit. Check user's permission before allowing the action. Signon screen reads the security table and passes the permission to the controller. Actions are selected by the global variable, next action.

Test the PF keys: 1 = menu, 10 = exit. PF2 is checked by the previously called module; if PF2 is set, the module sets next action to a default that varies according to the menu option.

Parameters:

action class	input	alpha 1	values: T = user has selected to manipulate tables only; may do security update, club table update, or view competition status and menu. C = user is managing a competition; may do any action on menu
permission	input	alpha 1	value from security table

Global variables:

next action (U)

Detailed Specifications:

```
DO while next action not = "E"
     IF next action = "M" or " "
          call menu program
     ELSE
          CHECK PERMISSION (next action, permission, accept)
                                                  /* SUBROUTINE
          IF accept = "N"
               output "ACTION NOT PERMITTED TO THIS
               OPEERATOR"
               call menu program
          ELSE
          IF action class = "T"
               CASE next action:
                    for AL, UC, DC:  call program
                    other:
                         output
                              "ONLY TABLE ACTIONS
                                        ARE PERMITTED"
                              "PRESS ENTER TO CONTINUE"
                         input enter
                         call menu program
               end of case
          ELSE      /* action class = "C"
               CASE next action:
                    for each value on menu, call program
                    other:
                         output
                              "INVALID CHOICE"
                              "PRESS ENTER TO CONTINUE"
                         input enter
                         call menu program
               end of case

     IF PF1
          next action = "M"
     IF PF10
          next action = "E"

end of DO loop

return
```

RF004 Signon Screen (OK, permission)

Purpose:

Input user ID and password. If the password is followed by a slash, input new password.

Read security file and load into an array of MAX SECURITY (15) records. The complete file is loaded in case the user later selects the security update actions. Place count of security records in competition rec.N security.

If the user ID is not found in the table, or if the given password is not the one entered in the table, set the OK flag to "N" and return.

If the user is permitted to use the system, save the new password if given. Output a message "NEW PASSWORD IN EFFECT". Resave the security file. Set the OK flag to "Y" and return.

If the security file is not found, display a message, set OK flag to "N", and return.

If the security file is found, but the user ID is not in it, or the password is incorrect, allow the user to try twice more before returning to the calling program, which will exit the system.

Parameters:

OK	output	alpha 1	"Y" = accept user "N" = reject user
permission	output	alpha 1	User's permission from security table. values: R, U, S.

Global variables:

security table (U)
competition rec.N security (U)

Detailed Specifications:

```
initialize security table by setting all user id's to blank
output screen title
OK = "N"
read security file into table
count number of security entries in N security
IF security table is not found
     output "SECURITY TABLE NOT FOUND"
     return

N tries = 0
REPEAT until N tries = 3 OR OK = "Y"
     add 1 to N tries
     output prompt for user ID
     input user ID
     output prompt foir password
     input password, slash, new password

     found = false
     sub = 0
     REPEAT until found = true
               OR user ID (sub) = " "
               OR sub = MAX SECURITY
          add 1 to sub
          IF user ID (sub) = user ID
               found = true
               IF password (sub) = password
                    IF slash = "/"
                         password (sub) = new password
                         output "NEW PASSWORD IN EFFECT"
                    parameter permission = permission (sub)
                    OK = "Y"
     end of repeat #2

     IF OK = "N"
          output "INVALID ID OR PASSWORD"
end of repeat #1

IF OK = "Y" and slash = "/"
     /* save updated security file
     SAVE NEW TABLE ("ST")          /* SUBROUTINE

return
```

RG001 Register Fencers

Default Action: O - Order Fencers

Purpose:

Input data for fencers and save in registration table. Do not permit registration once competition has begun.

Parameters:

none

Global variables:

event number
name indicator (event) (U)
N competing (event) (U)
registration table (U)

Detailed Specifications:

```
IF N rounds pools (event) not = 0
   or
   N rounds elim (event) not = 0
          output "COMPETITION HAS BEGUN.  REGISTRATION IS
                  CLOSED"
          next action = "M"
          return

count = 0       /* counts fencers registered since last save
DO WHILE next action = "R"    /* "R" = register fencers
     REGISTER ONE FENCER (count)          /* SUBROUTINE
end

name indicator (event) = "N"  /* "N" = not alphabetized
N competing (event) = N reg (event)
IF count > 0
     SAVE NEW TABLE ("RT")                /* SUBROUTINE
     SAVE NEW TABLE ("CR")                /* SUBROUTINE

/* next action is input by subroutine Register One Fencer
return
```

RG020 Register One Fencer (count)

Purpose:

Get and store values in registration record for one fencer. Set fencer's status to competing. Save registration table periodically. Increment count of number registered. Do not permit more than MAX FENCERS (175) to be registered. User may enter "R" or blank to keep registering fencers.

Save point is a constant for count of fencers to register before saving the registration file.

Parameters:

count	output	intgr 2	count of fencers registered since the registration files were saved

Global variables:

registration table
next action

Detailed Specifications:

```
id = N reg (event) + 1
IF id > MAX FENCERS
     output "ALREADY ", MAX FENCERS, " REGISTERED"
     output "THIS FENCER REJECTED"
     next action = "M"
     return

REGISTRATION SCREEN                    /* SUBROUTINE
     (id, name, club 1, club 2, group 1, group 2,
      USFA rank, position)
store values of name, club 1, club 2, group 1, group 2,
      USFA rank, position in registration record (id)
fencer.status (id) = "C"
N reg (event) = id
add 1 to count       /* increment count of fencers registered
                     /*  since last save
IF count = SAVE POINT
     /* save registration table and competitiion file
     SAVE NEW TABLE ("RT")          /* SUBROUTINE
     SAVE NEW TABLE ("CR")          /* SUBROUTINE
     count = 0

output "NEXT ACTION: "
input next action
IF PF2
     next action = "M"
IF next action = " "
     next action = "R"

 return
```

OR002 Rank Fencers

Default Action: AP - Assign Pools

Purpose:

Set rank indicator to "Y". Do not sort if the list is already sorted.

Parameters:

none

Global variables:

event number
N reg (event)
rank indicator (event)
rank tree root (event)
registration table

Detailed Specifications:

```
output screen headers

IF rank indicator (event) = "Y"
     output "FENCERS ARE ALREADY RANKED"
     output "NEXT ACTION: "
     input next action
     IF PF2
          next action = "AP"
     return

rank tree root (event) = NIL PTR
number eliminated = 0
number withdrawn = 0

IF N rounds pools (event) = 0
   and
   N rounds elim (event) = 0
          /* this is the first time ranking was requested
          RANK ON USFA CLASS                    /* SUBROUTINE
          number ranked = N reg (event)
ELSE
          check for all results input          /*  see below
          RANK ON INDICATORS                    /* SUBROUTINE
            (number ranked, number eliminated,
             number withdrawn)

rank indicator (event) = "Y"
output "NUMBER OF FENCERS RANKED "  number ranked
output "NUMBER OF FENCERS ELIMINATED AFTER LAST ROUND "
        number eliminated
IF number withdrawn > 0
     output "NUMBER OF FENCERS WITHDRAWN "
             number withdrawn
IF N rounds elim (event) = 0
     output "NUMBER OF FENCERS COMPETING IN NEXT ROUND "
             N competing (event)

output "NEXT ACTION: "
input next action
IF PF2
     next action = "AP"

 return
```

Check For All Results Input:

```
IF N rounds elim (event) = 0
     /* previous round was pools
     IF any pool has outstanding = "T"
          output "POOL " n " RESULTS STILL NOT INPUT"
          output "RANKING NOT PERFORMED"
          return
     ELSE
          no action
ELSE
     IF N elim outstanding (event) not = zero
          output "RESULTS NOT INPUT FOR ALL ELIMINATION
                  BOUTS "
          output "RANKING NOT PERFORMED"
          return
```

OR010 Rank on USFA Class

Purpose:

Prepare for first match assignments. Build rank tree in ascending order by USFA class for all registered fencers. Set rank numbers so that first A = 1 and last U = N-reg (event). Fencers with point position precede those with no point position. Lowest point position is rank number 1.

Parameters:

none

Global variables:

registration tabale

Detailed Specifications:

```
output "RANKING ALL FENCERS ON USFA CLASS"

For id = 1 to N reg (event)
     INSERT IN RANK TREE                    /* SUBROUTINE
          (id, rank tree root (event))
end

count = 0
UPDATE RANKS ASCENDING ORDER                /* SUBROUTINE
          (count, rank tree root (event))

return
```

RP001 Review Pools

Default Action: PP - Print Pools

Purpose:

Display pool assignments. Allow user to change the composition of the pools after initial assignment and before forms are printed.

For each pool, display maximum number of slots (B or D, whichever is greater). For each fencer, display name (20 chars), club code (4 chars), USFA class (1 char), rank in the competition (3 chars). Fencers of class A with point position have class A*, so that operator will know that they can not be moved.

Allow operator to move fencers around in pools as long as the final result is A pools of B fencers and C pools of D fencers, and as long as there are never more than MAX IN POOL fencers in a pool. One fencer may be held while others are swapped or moved. Program does not permit fencers with point position to be moved.

The operator may:

A = approve	PF1
H = hold fencer	PF3
I = insert fencer on hold into a pool	PF4
M = move from one pool to another	PF5
S = swap two fencers	PF6
F = page forward one screen	PF7
B = page back one screen	PF8
G = go to a pool	PF9

Option G prompts operator for the pool number to be displayed at the top left position of the screen.

When multiple screens are required to show all pools and there are N pools on each screen, the last screen should also have N pools on it, with the last pool in the bottom right corner.

Parameters:

none

Global variables:

event number
event rec.B (event)
event rec.D (event)
N pools (event)

Detailed Specifications:

```
constant: LINES AVAIL = 21

/* Plan screen layout. There are 21 lines avaiable on which
/* to print fencers and pool headers.

/* Get length of block for 2 pools side by side
IF event rec.B > event rec.D
     pool size = B
ELSE
     pool size = D
block size = pool size + 1

/* Get number of pools that will fit on the screen
screen blocks = LINES AVAIL DIV pool size
screen pools = 2 * screen blocks
                              /* number of pools on screen
blank lines = LINES AVAIL - (screen blocks * block size)

/* Initialize
first pool = 1
last pool = screen pools
hold ID = 0
approved = "N"

REPEAT until approved = "Y"
     RP SCREEN                          /* SUBROUTINE
          (first pool, last pool, pool size, hold ID,
           LINES AVAIL)
     execute wait loop until key is pressed
     CASE:
          PF1: APPROVE (approved)  /* SUBROUTINE
          PF3: HOLD                /* SUBROUTINE
                    (hold ID, first pool, last pool,
                     blank lines, block size)
          PF4: INSERT              /* SUBROUTINE
                    (hold ID, first pool, last pool,
                     blank lines, block size)
          PF5: MOVE                /* SUBROUTINE
                    (first pool, last pool,
                     blank lines, block size)
          PF6: SWAP                /* SUBROUTINE
                    (first pool, last pool, blank lines,
                     block size)
          PF7: FORWARD             /* SUBROUTINE
                    (screen pools, first pool, last pool,
                     pool size, hold ID, blank lines)
          PF8: BACK                /* SUBROUTINE
                    (screen pools, first pool, last pool,
                     pool size, hold ID, blank lines)
```

```
          PF9: GO TO                    /* SUBROUTINE
                              (screen pools, first pool, last pool,
                               pool size, hold ID, blank lines)
          other: no action - drop through
     end  /* of CASE
end /* of REPEAT

output screen 2:
     "POOLS APPROVED"
     "ENTER PF2 TO PRINT POOL SHEETS"
     "OR ENTER NEXT ACTION: "

input next action
IF PF2
     next action = "PP"

 return
```

RP010 Review Pools Screen (first pool, last pool, pool size, hold ID, blank lines)

Purpose:

Display pools for review.

Parameters:

first pool	input	intgr 2	pool number of first pool on this screen
last pool	input	intgr 2	pool number of last pool on this screen
pool size	input	intgr 1	length for block of pools
hold ID	input	intgr 3	ID of fencer on hold
blank lines	input	intgr 1	number of lines available to be left blank

Global variables:

event number (R)
N pools (event) (R)
pool table (R)
registration table (R)

Detailed Specifications:

```
clear screen
output top line with event rec.N pools (event)
IF blank lines GE 3
     output blank line after top line

pool = first pool
REPEAT until pool GE last pool
     pool 1 = pool
     pool 2 = pool + 1
     output pool header for pool 1
     IF pool 2 LE last pool
          output pool header for pool 2
     row = 0
     REPEAT until row = pool size
          row = row + 1
          IF row LE pool rec.pool size (pool 1)
               reg ptr = pool rec.id (row, pool 1)
               output on left:
                    row
                    fencer.name (reg ptr)
                    fencer.USFA class (reg ptr)
                    IF fencer.point position (reg ptr)
                         not = zero
                              output "*"
                    fencer.rank (reg ptr)
                    fencer.club group 1 (reg ptr)
                    fencer.club group 2 (reg ptr)
          IF row LE pool rec.pool size (pool 2)
               reg ptr = fencer.id (pool 2)
               output on right:
                    row
                    fencer.name (reg ptr)
                    fencer.USFA class (reg ptr)
                    IF fencer.point position (reg ptr)
                         not = zero
                              output "*"
                    fencer.rank (reg ptr)
                    fencer.club group 1 (reg ptr)
                    fencer.club group 2 (reg ptr)
          output end of line       /* blank line if no
                                   /* output for either pool
     end of REPEAT 2
     pool = pool + 2
end of REPEAT 1

IF blank lines GE 1
     output blank line
```

```
/* output fencer on hold
IF hold ID = 0
     output blank line
ELSE
     output fencer.name (hold ID)
            fencer.USFA class (hold ID)
            IF fencer.point position (hold ID) not = zero
                 output "*"
            fencer.rank (hold ID)
            fencer.club group 1 (hold ID)
            fencer.club group 2 (hold ID)

IF blank lines GE 2
     output blank line
output bottom line of sreen

return
```

UC001 Update Clubs

Default Action: M - Menu

Purpose:

Build or update the club table.

Operator may add, delete, or update club records, display the club table, or print the club table.

(Subroutine Show Club Table lists the entire table if the passed parameter is blank.)

Parameters:

none

Global variables:

club table
sorted club names array
sorted club codes array

Detailed Specifications:

```
update flag = "N"
update action = " "

REPEAT until update action = "E"
     output update menu

     valid = "N"
     REPEAT until valid = "Y"
          input update action
          CASE: action
               A,C,D,L,P,E: valid = "Y"
               other:
                    output "INVALID CHOICE"
     end of repeat

     IF update action = "C" or D"
          output "ENTER CLUB CODE: "
          input club code
          update flag = "Y"

     CASE: update action
          A:   ADD CLUBS                          /* SUBROUTINE
          C:   CHANGE CLUBS (club code)           /* SUBROUTINE
          D:   DELETE CLUB (club code)            /* SUBROUTINE
          L:   SHOW CLUB TABLE (" ")              /* SUBROUTINE
          P:   PRINT CLUB TABLE                   /* SUBROUTINE

end of repeat

IF update flag = "Y"
     SAVE NEW TABLE ("CT")

output "NEXT ACTION: "
input next action
IF PF2
     next action = "M"

 return
```

UC002 Add Clubs

Purpose:

Input code for club to be added. Add clubs until input club code is XXXX. Refuse to add if code is already in the club table.

Pass club code and club name to subroutine for addition to club table and entry into the sorted club arrays.

Parameters:

none

Global variables:

club table (U)
club arrays (U)
competition rec.N clubs (U)

Detailed Specifications:

```
clear screen
output screen header "(UC)                ADD CLUB"
output "ENTER XXXX IN CLUB CODE TO STOP"

club = "    "
REPEAT until club = "XXXX"

     output "CODE: "
     input club
     output "NAME: "
     input name
     output "GROUP: "
     input group

     output "CORRECT? "
     input response

     IF response = "Y"
          found = false
          subscript = 0
          SEARCH CLUB CODES                    /* SUBROUTINE
                    (club, found, subscript)
          IF found = true
               ADD TO CLUB TABLE               /* SUBROUTINE
                    (club, name, group, found, subscript)
          ELSE
               output "CLUB CODE ALREADY IS IN TABLE FOR  "
                       club rec.club name (subscript)

end of repeat

return
```

RG060 Add To Club Table (code, name, group, found, subscript1)

Purpose:

Enter code, name, and group into a new record appended to club table. Increment count of clubs.

Enter code into sorted club codes array and put pointer into the new club table record. Move succeeding codes down in the code array. Enter name in the sorted club names array, and put pointer into the new club table record. Move succeeding entries down in the name array. Update pointers in the club table for records that were moved down in the sorted arrays.

ID's in the club table do not change because the add is done at the end of the table.

Refuse to add club if the name is already in the sorted name array. The code has already been checked and is not present in the code array.

Parameters:

code	input	alpha 4	club code to be added
name	input	alpha 20	name of club to be added
group	input	alpha 4	affinity group of club to be added
found	output	alpha 1	Y = name already is in name table; add refused N = name not found in table; add accepted
subscript1	input	intgr 3	subscript in sorted club code table preceding point for add

Global variables:

club table (U)
sorted club arrays (U)
N clubs (U)

Detailed Specifications:

```
/* get subscript of place for add in sorted club names array
SEARCH CLUB NAMES (name, found, subscript2)  /* SUBROUTINE

IF found = "Y"
     /*  club name is already in the table
     id = sorted club names.id (subscript2)
     output "ALREADY IN TABLE. CODE IS ",
             club table.code (id)
ELSE      /* found = "N"
     /* increment N clubs - this is the ID for the new club
     add 1 to N clubs
     club name (N clubs) = name
     club code (N clubs) = code
     affinity group (N clubs) = group

     /* insert code into sorted club codes array
     FOR items (I) in sorted club codes
               from N clubs down to subscript1   + 2
          /* move I-1 to I
          sorted club code (I) = sorted club code (I-1)
          ptr = sorted club code (I)
          /* update code pointer in club table for record
          /*      that was moved
          club table.code ptr (ptr) = I
          /* last move puts subscript1 + 1 into
          /* subscript1 + 2
     end of for

     ptr = subscript1 + 1          /* add new code at ptr
     sorted club codes (ptr) = N clubs
     club table.code ptr (N clubs) = ptr

     /* insert name into sorted club names array
     FOR items (I) in sorted club names
               from N clubs down to subscript2   + 2
          /* move I-1 to I
          sorted club name (I) = sorted club name (I-1)
          ptr = sorted club name (I)
          /* update name pointer in club table for record
          /*   that was moved
          club table.name ptr (ptr) = I
          /* last move puts subscript2 + 1 into
          /*   subscript2 + 2
     end of for

     ptr = subscript1 + 1          /* add new name at ptr
     sorted club names (ptr) = N clubs
     club table.name ptr (N clubs) = ptr

return
```

SS001 Security Menu

Default Action: M - Menu

Purpose:

Provide menu and control for maintaining security data. Add operators to system with permissions. Delete Operators. List operators, permissions and passwords. Display one operator's name, permission, and password.

Operator data can not be changed. An operator must be deleted and re-added if a field is incorrect. User can change his/her password at signon.

Valid permissions are:
R = display only
U = update
S = security update

Operator with R permission is denied use of: AE, AP, CR, IE, IP, O, R, RP, UC, and W screens.

Operator with U has access to all the screens except the security update (AL).

Operator with S permission may access all screens.

If table was updated, the new table is saved.

Security menu does not display the screen name.

Global variables:

Security table
N_security in Competition control record
next action

Detailed Specification:

```
/* initialization
error = no          /* indicates that a choice was not valid
update = no         /* flag to control saving tables
choice = " "

REPEAT until choice = "E"
     If error = no
          clear screen
          display menu
          /* get new choice
          valid = no
          REPEAT until valid = yes
               output "CHOICE: "
               input choice
               IF choice = "A" or "D" or "E" or "L" or "Q"
                    valid = yes
          END of repeat
     END IF
     error = no          /* reset error flag
     IF choice = "D" or "Q"
          /* get operator ID
          output "OPERATOR ID:"
          input id
          /* find ID in security table
          found = no
          I = 0
          REPEAT until found = yes or I = N_security
               add 1 to I
               IF operator.id (I) = id
                    found = yes
          END of repeat
          IF found = no
               output "OPERATOR NOT ON FILE"
               error = yes
          END IF
     END IF
     IF error = no
          IF choice = "A" or "D"
               update = yes
          END IF
          case choice
               A:   add operators              /* SUBROUTINE
               D:   delete operator (I)        /* SUBROUTINE
               L:   list operators             /* SUBROUTINE
               Q:   query operator (I)         /* SUBROUTINE
END of repeat
```

```
IF update = yes
     save new table ("ST")                          /* SUBROUTINE
output "NEXT ACTION:"
input next action
IF PF2
     next action = "M"
return
```

TE001 Transfer Event

Default Action: M - Menu

Purpose:

Permit user to stop processing files for the current event and get files for a different event and process them. Refuse request if the event is not open. Files are saved by the modules that update them; the competition table (competition and event control records) are saved at exit. So the current files do not have to be saved.

Load the registration file, the pool table, and the elimination list.

Prompt the operator for the event number, and inform operator of the name of the event that has been transferred to.

Replace the global variable for event number with the new value.

Parameters:

none

Global variables:

event number (U)

Detailed Specifications:

```
output "CURRENT EVENT IS " event number ":"
        event rec.title L (event number)

valid = "N"
REPEAT until valid = "Y"
     output "INPUT NEXT EVENT NUMBER:"
     input next event
     validate event number       /*  see below
     IF event number is valid
          valid = "Y"
          event number = next event
end of repeat

/* control B loads registration table, pool table, /*
elimination lists
LOAD TABLES (B)           /* SUBROUTINE

output 'CURRENT EVENT IS NOW " event number ":"
        event rec.title L (event number)

output "NEXT ACTION: "
input next action
IF PF2
     next action = "M"

 return



validate event number:

     IF next event GE 1 and LE 4
          IF event rec.status (next event) = "O"
               valid = "Y"
          ELSE
               output "EVENT " next event
                      event rec.title L (next event)
                      " IS CLOSED. NO TRANSFER."
     ELSE
          output "MUST BE 1 - 4. NO TRANSFER."
```

APPENDIX F

OBJECTS AND OPERATIONS FOR ADA DESIGN

READY FENCE!

Object-Oriented design

DEFINE THE PROBLEM

Permit an OPERATOR to select functions to conduct a COMPETITION. For each EVENT in the competition, input FENCERS and add to a REGISTRATION list. Rank the fencers in the list, assign them to bouts in POOLS or ELIMINATION. Permit the operator to adjust the POOL TABLE of assignments to avoid matching fencers from the same fencing CLUB. Input results of previous rounds and update the statistics for each fencer. Maintain a TABLE OF FENCING CLUBS to which the competitors belong. Also maintain a SECURITY TABLE of operators permitted to use the system. Print SCORE SHEETS and REPORTS.

IDENTIFY THE OBJECTS AND THEIR ATTRIBUTES

COMPETITION	Abstract state machine. A single object containing the state of the competition and the state of each event.
EVENT	Abstract data type with collection of operations.
FENCER	Abstract data type. The central abstraction.
POOL	Abstract data type.
POOL TABLE	Abstract state machine, a data store.
ELIMINATION LIST	Abstract state machine, a data store.
REGISTRATION	Abstract state machine, a data store.
CLUB	Abstract data type.
CLUB TABLE	Abstract state machine, a data store.
OPERATOR	Abstract data type.
SECURITY TABLE	Abstract state machine, a data store.
SCORE SHEETS	Program related units.
REPORTS	Program related units.

IDENTIFY THE OPERATIONS ON THE OBJECTS

COMPETITION

open competition	close competition
load competition	save competition
display competition	
initialize current event	
change current event	
close current event	
perform function	
get report data	
is open	

EVENTS

open event	close event
load event	save event
assign elimination	
assign pools	
correct registration	
display elimination assignments	
display pool assignments	
delete fencer	
input elimination results	
input pool results	
order registration list	
register fencers	
review pools	
withdraw fencer	
get values	

FENCERS

REGISTRATION

load registration table	save registration table
initialize registration table	
add fencer	
delete fencer	
modify fencer	
get fencer	
build name index	build rank index
read by name	read by rank

POOLS

insert fencer	
remove fencer	
get from pool	
pool size	

POOL TABLE

load pool table	save pool table
initialize pool table	

ELIMINATION LISTS

- load lists — save lists
- initialize lists
- insert fencer
- remove fencer
- record winner
- record loser
- get elim
- get winner
- get loser

CLUBS

CLUB TABLE

- load club table
- save club table
- in club table
- club menu
- add club
- delete club
- modify club
- display club table
- print club report
- search club names
- search club codes

OPERATORS

SECURITY TABLE

- load security table — save security table
- security menu
- add operator
- modify operator
- delete operator
- validate operator
- operator inquiry
- display security table
- password change by operator

SCORE SHEETS

- print pool form
- print elimination form

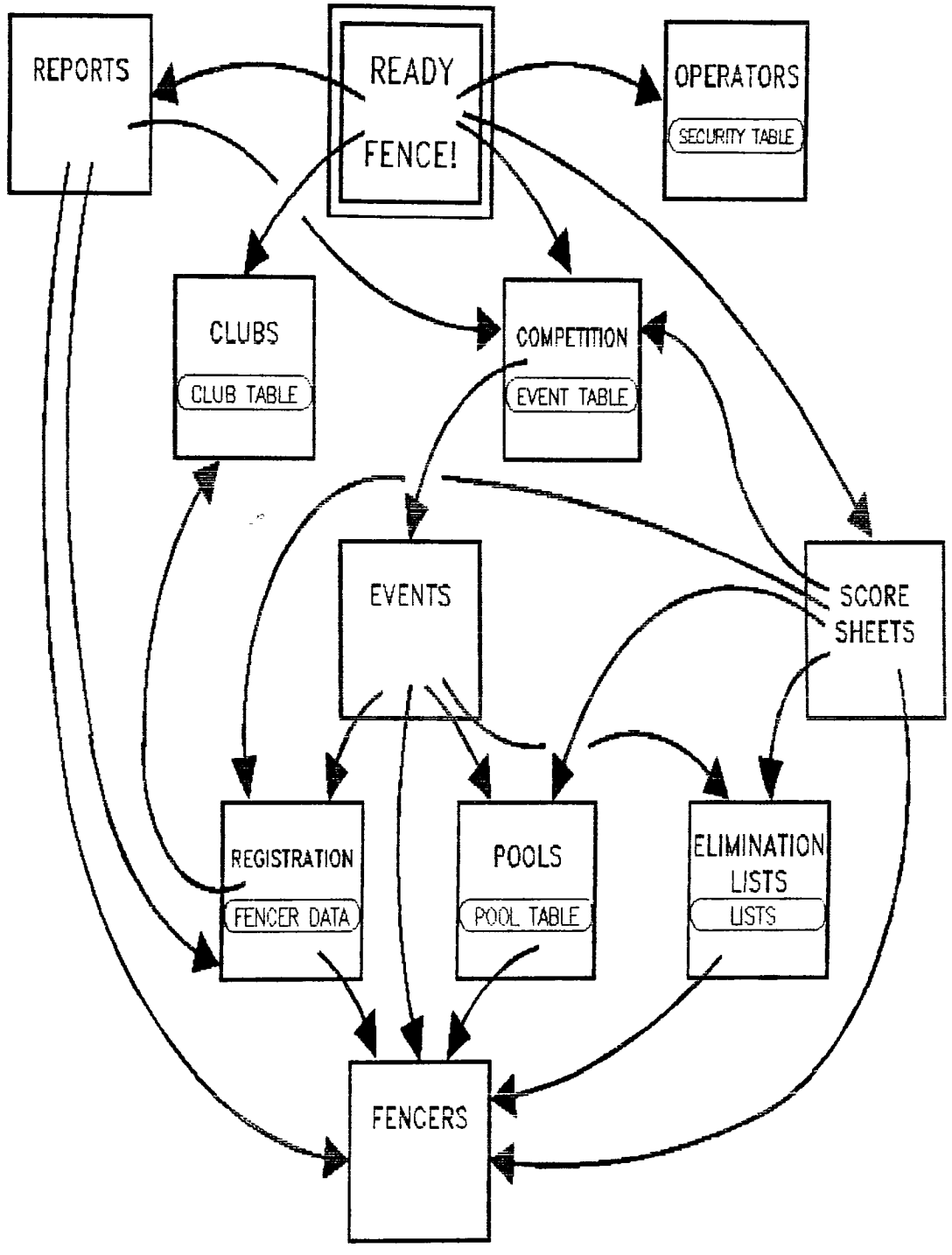
REPORTS

- specify report
- print report by name
- print report by rank
- print registration report by ID

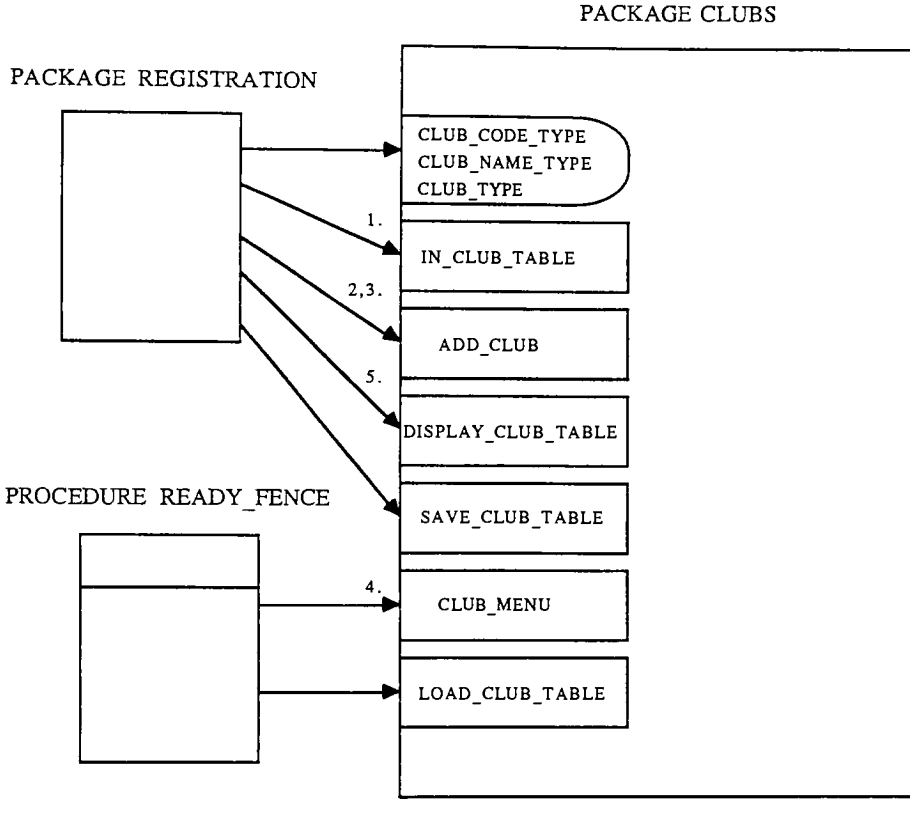
APPENDIX G

GRAPHICAL REPRESENTATION FOR ADA DESIGN

ESTABLISH THE VISIBILITY OF EACH OBJECT IN RELATION TO OTHER OBJEC

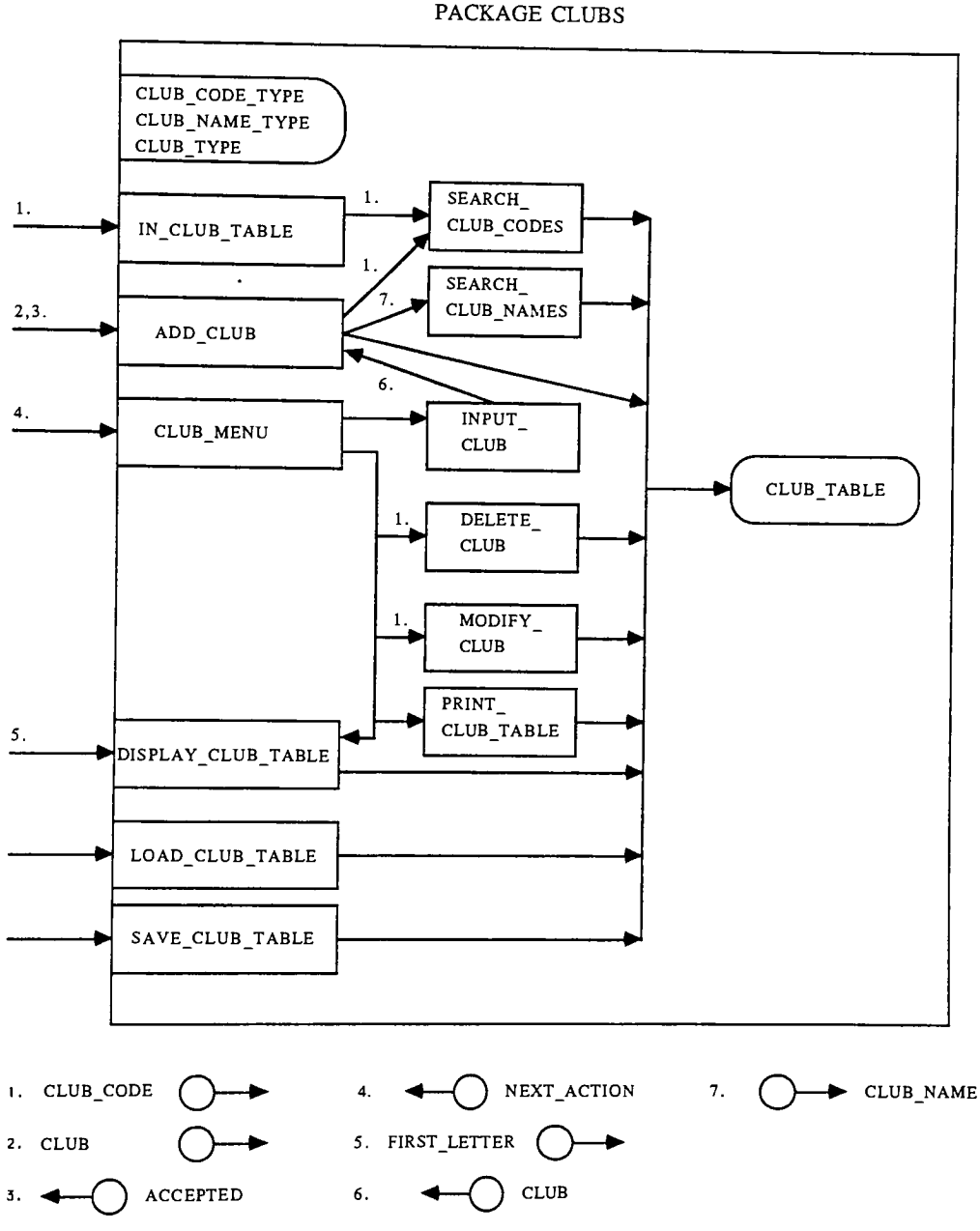


CLUBS EXTERNAL VIEW

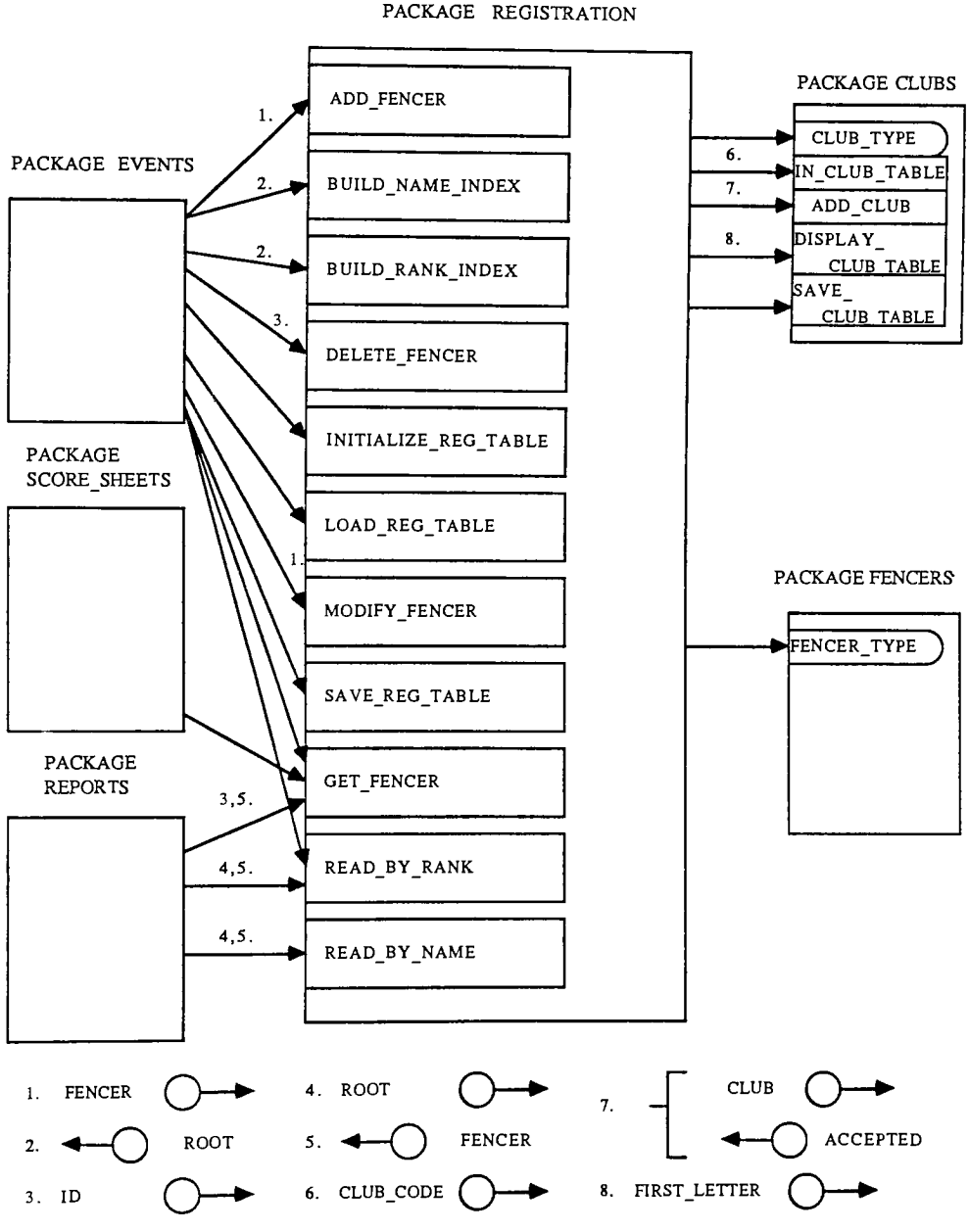


1. CLUB_CODE
2. CLUB
3. ACCEPTED
4. NEXT_ACTION
5. FIRST_LETTER

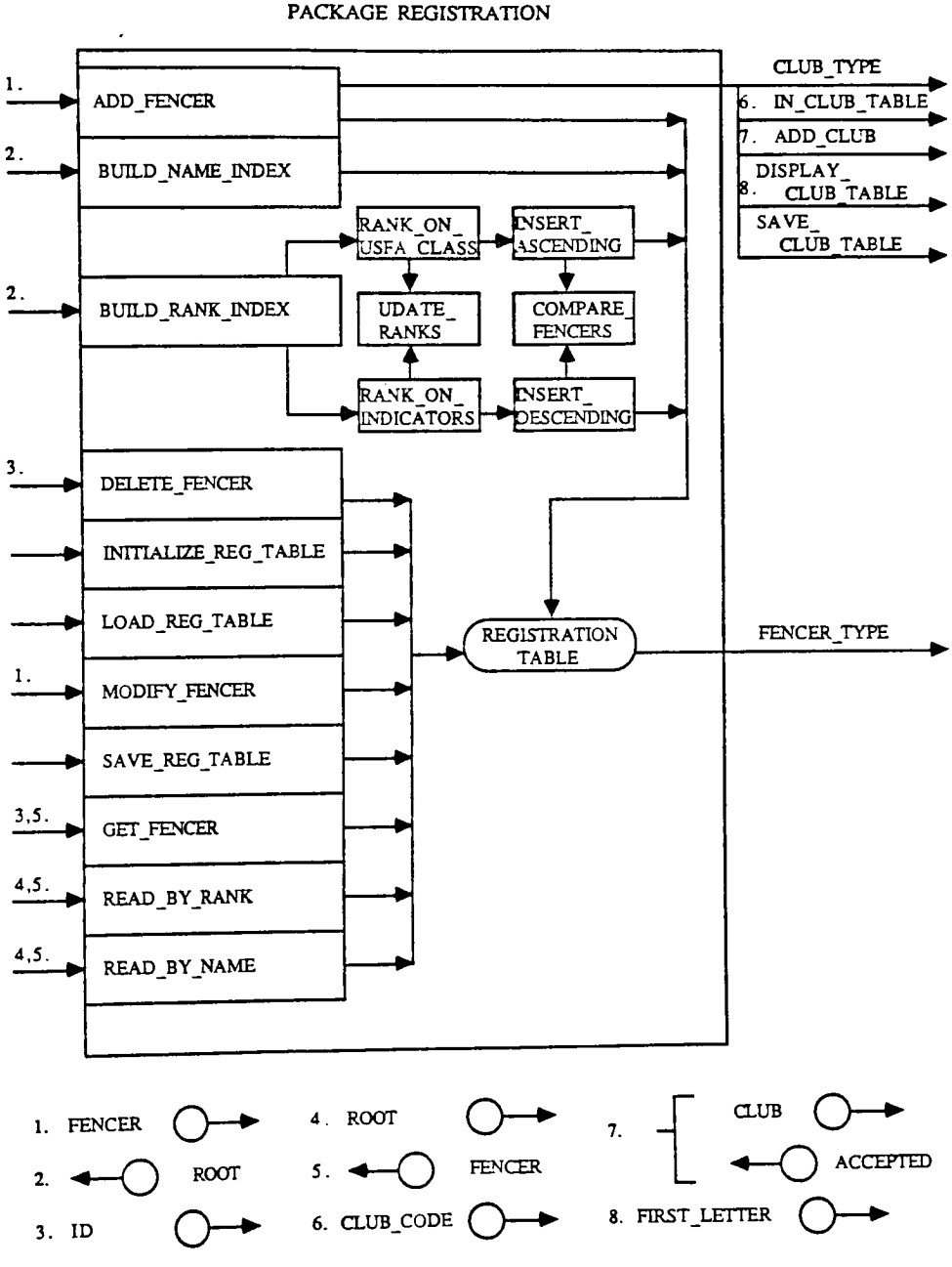
CLUBS INTERNAL STRUCTURE



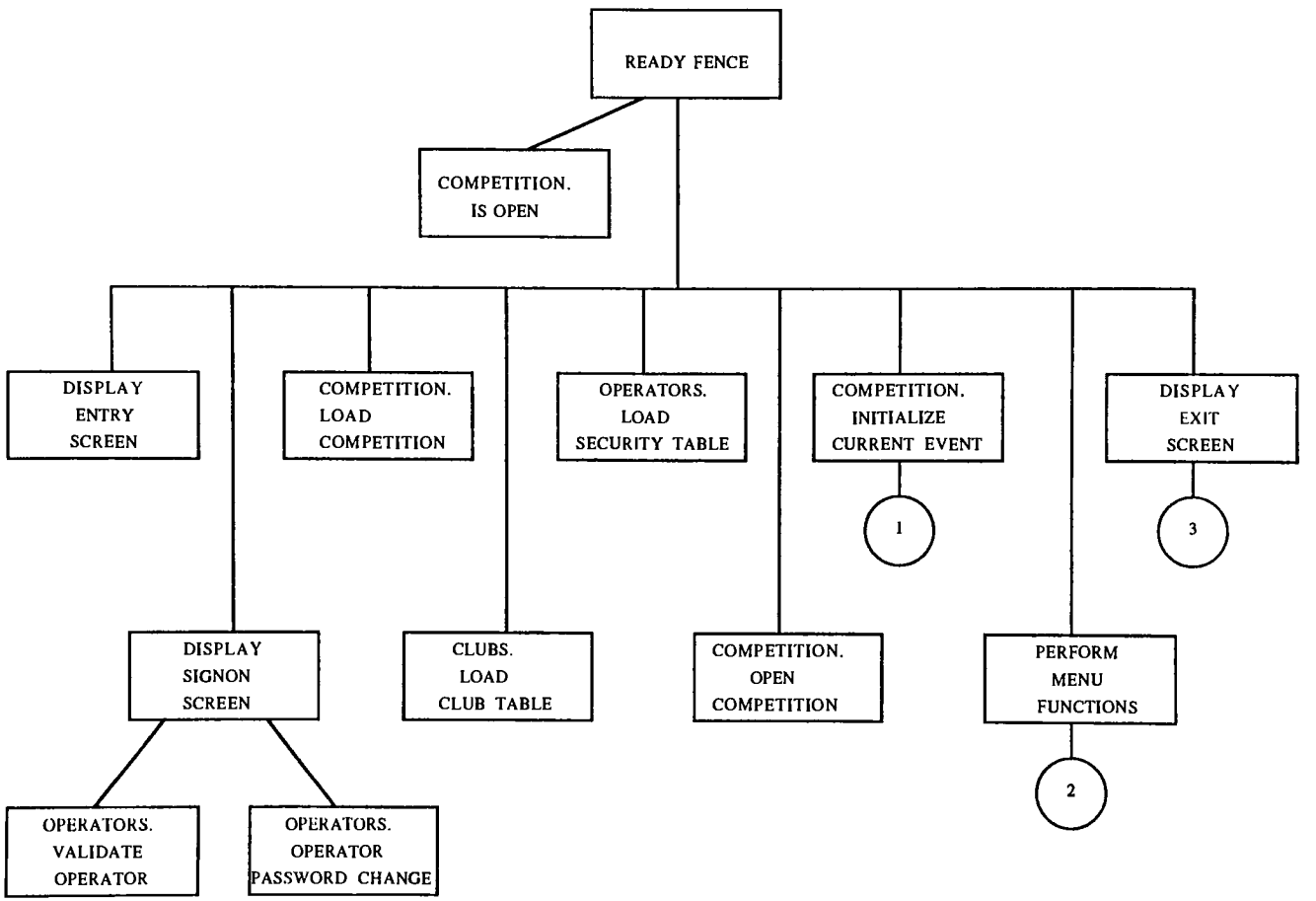
REGISTRATION EXTERNAL VIEW

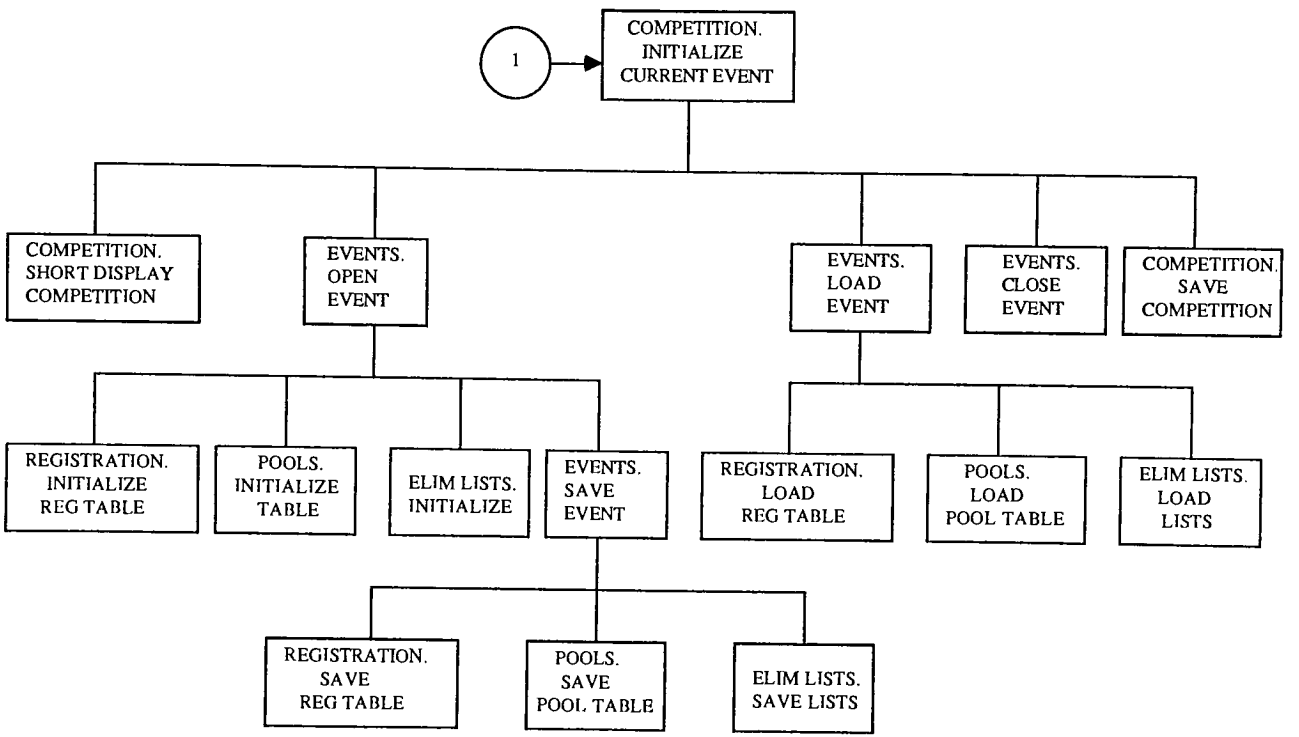


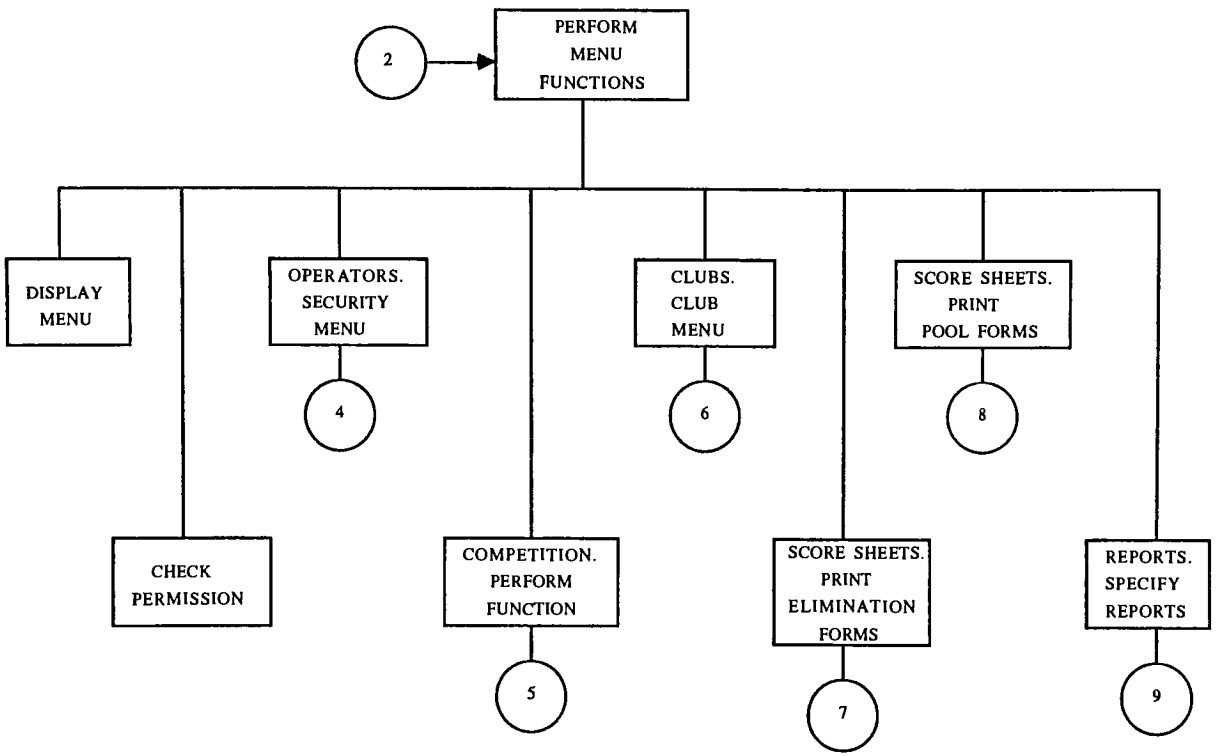
REGISTRATION INTERNAL STRUCTURE

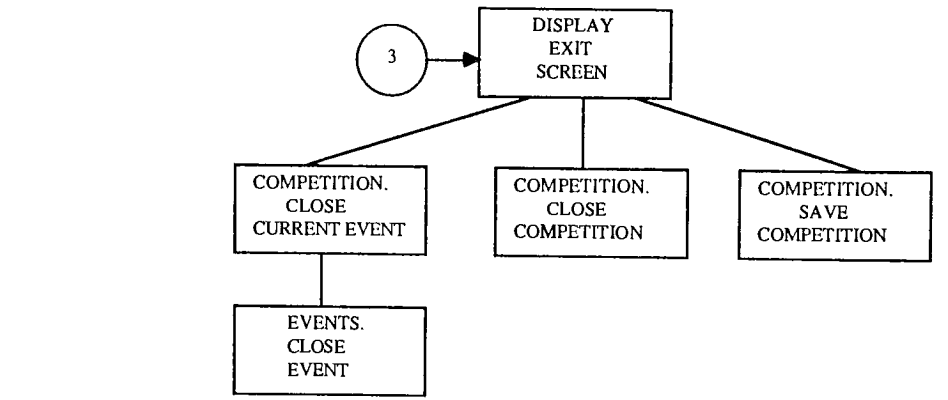


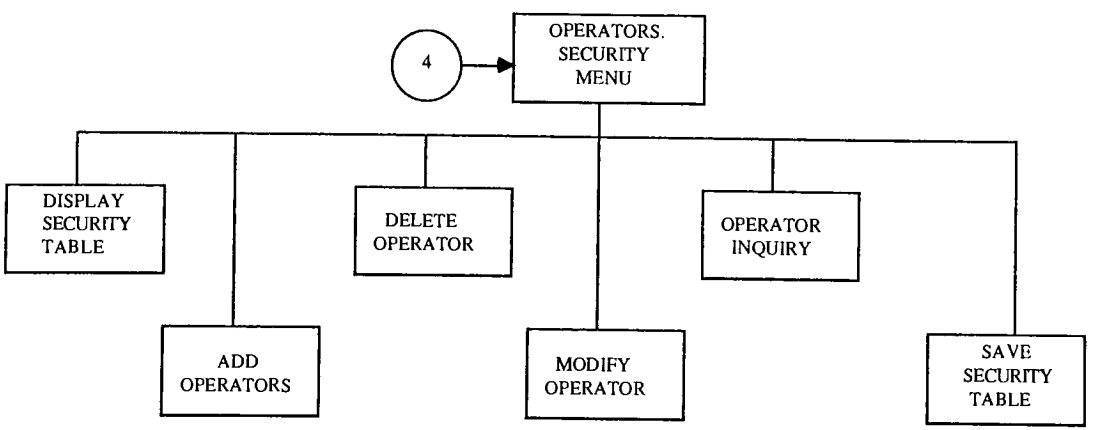
ADA READY FENCE! STRUCTURE

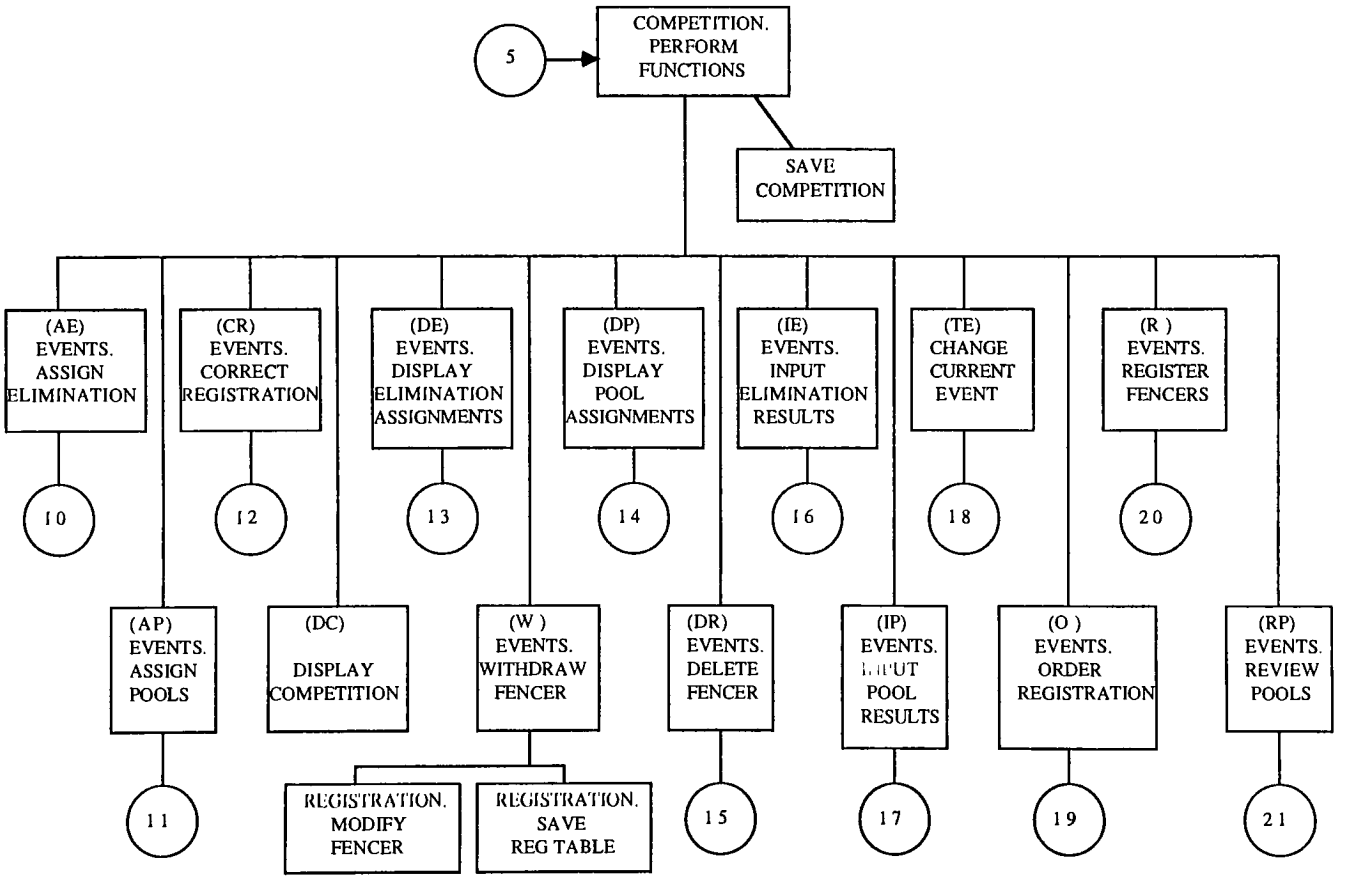


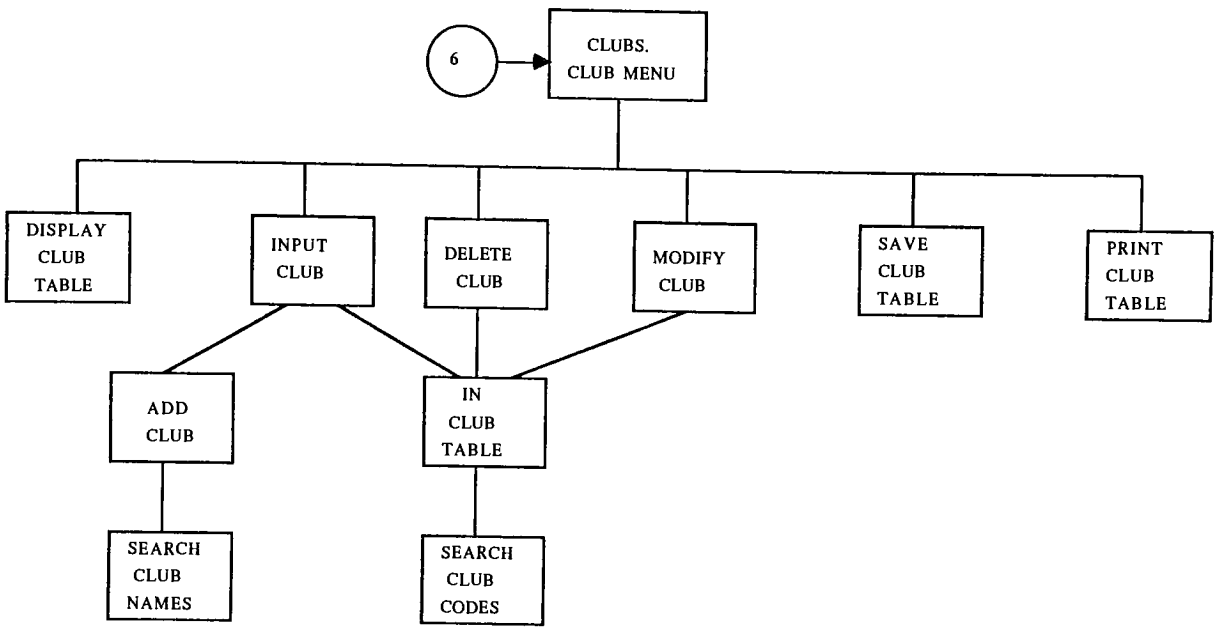

1
COMPETITION. INITIALIZE CURRENT EVENT
COMPETITION. SHORT DISPLAY COMPETITION
EVENTS. OPEN EVENT
EVENTS. LOAD EVENT
EVENTS. CLOSE EVENT
COMPETITION. SAVE COMPETITION
REGISTRATION. INITIALIZE REG TABLE
POOLS. INITIALIZE TABLE
ELIM LISTS. INITIALIZE
EVENTS. SAVE EVENT
REGISTRATION. LOAD REG TABLE
POOLS. LOAD POOL TABLE
ELIM LISTS. LOAD LISTS
REGISTRATION. SAVE REG TABLE
POOLS. SAVE POOL TABLE
ELIM LISTS. SAVE LISTS

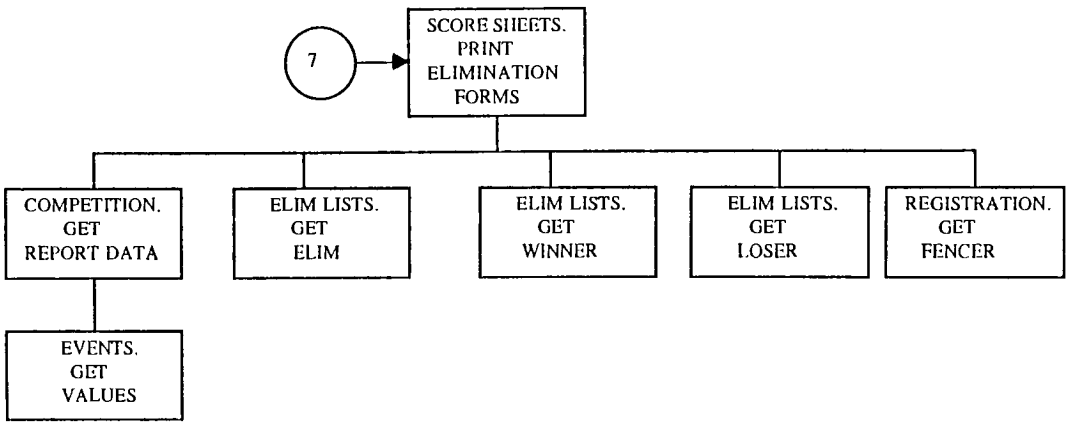

2
PERFORM MENU FUNCTIONS
DISPLAY MENU
CHECK PERMISSION
OPERATORS. SECURITY MENU
4
COMPETITION. PERFORM FUNCTION
5
CLUBS. CLUB MENU
6
SCORE SHEETS. PRINT ELIMINATION FORMS
7
SCORE SHEETS. PRINT POOL FORMS
8
REPORTS. SPECIFY REPORTS
9

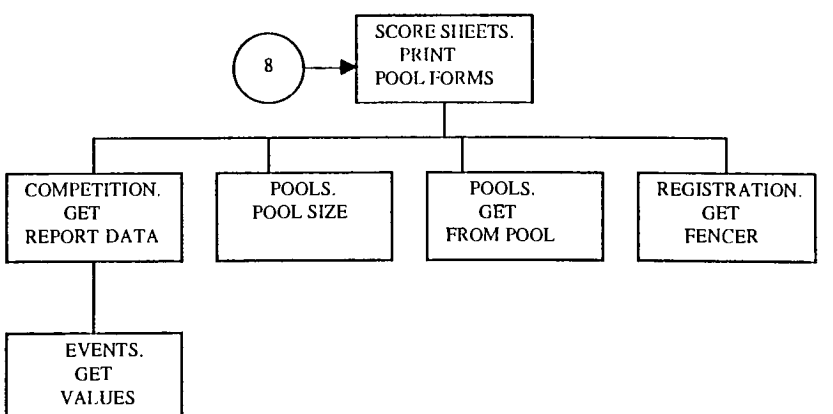

3
DISPLAY EXIT SCREEN
COMPETITION. CLOSE CURRENT EVENT
COMPETITION. CLOSE COMPETITION
COMPETITION. SAVE COMPETITION
EVENTS. CLOSE EVENT

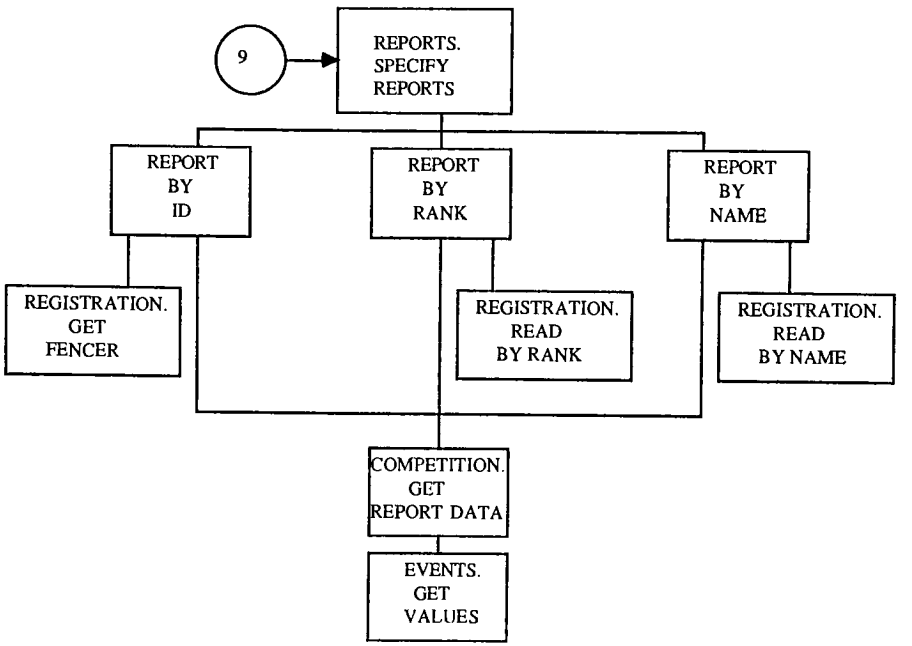

4
OPERATORS. SECURITY MENU
DISPLAY SECURITY TABLE
ADD OPERATORS
DELETE OPERATOR
MODIFY OPERATOR
OPERATOR INQUIRY
SAVE SECURITY TABLE

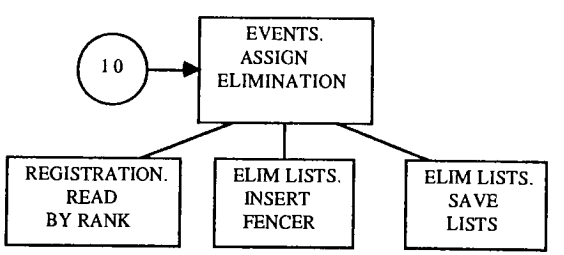

5
COMPETITION. PERFORM FUNCTIONS
SAVE COMPETITION
(AE) EVENTS. ASSIGN ELIMINATION
(CR) EVENTS. CORRECT REGISTRATION
(DE) EVENTS. DISPLAY ELIMINATION ASSIGNMENTS
(DP) EVENTS. DISPLAY POOL ASSIGNMENTS
(IE) EVENTS. INPUT ELIMINATION RESULTS
(TE) CHANGE CURRENT EVENT
(R) EVENTS. REGISTER FENCERS
10
12
13
14
16
18
20
(AP) EVENTS. ASSIGN POOLS
(DC) DISPLAY COMPETITION
(W) EVENTS. WITHDRAW FENCER
(DR) EVENTS. DELETE FENCER
(IP) EVENTS. INPUT POOL RESULTS
(O) EVENTS. ORDER REGISTRATION
(RP) EVENTS. REVIEW POOLS
11
REGISTRATION. MODIFY FENCER
REGISTRATION. SAVE REG TABLE
15
17
19
21

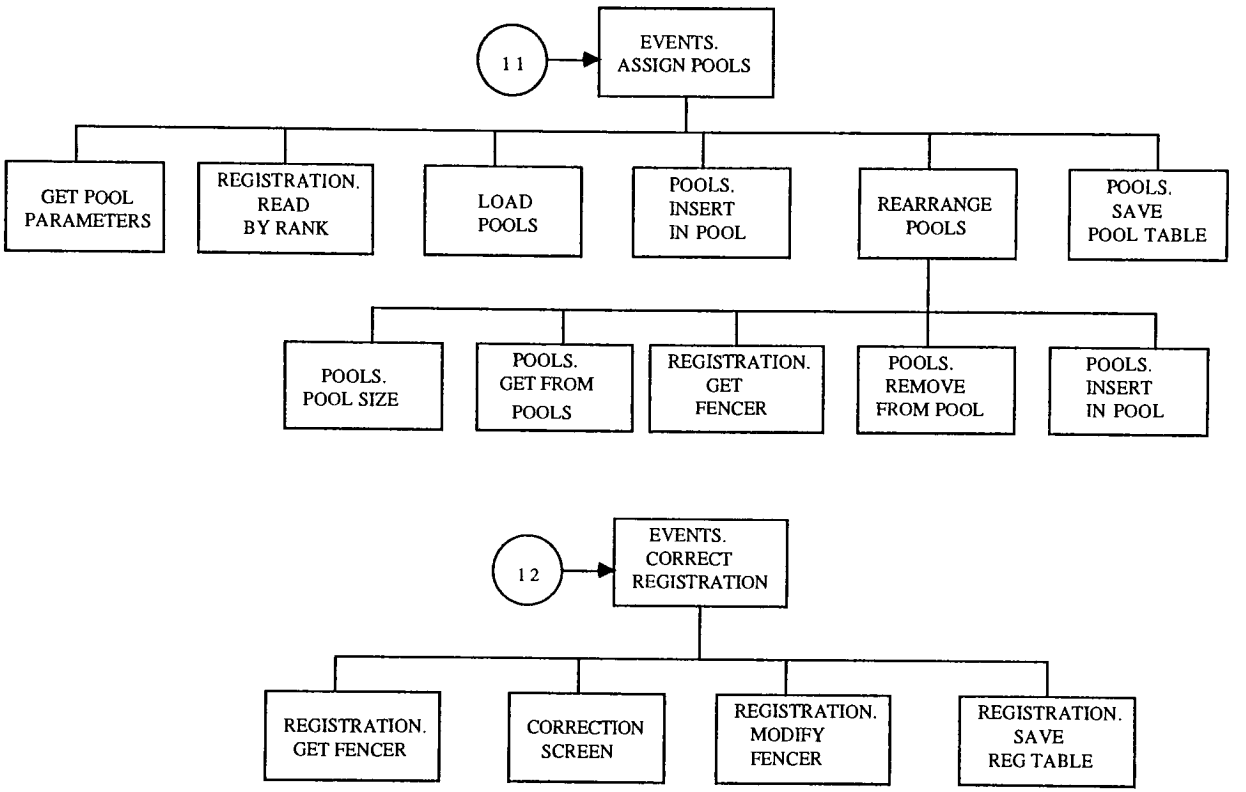

6
CLUBS.
CLUB MENU
DISPLAY
CLUB
TABLE
INPUT
CLUB
DELETE
CLUB
MODIFY
CLUB
SAVE
CLUB
TABLE
PRINT
CLUB
TABLE
ADD
CLUB
IN
CLUB
TABLE
SEARCH
CLUB
NAMES
SEARCH
CLUB
CODES

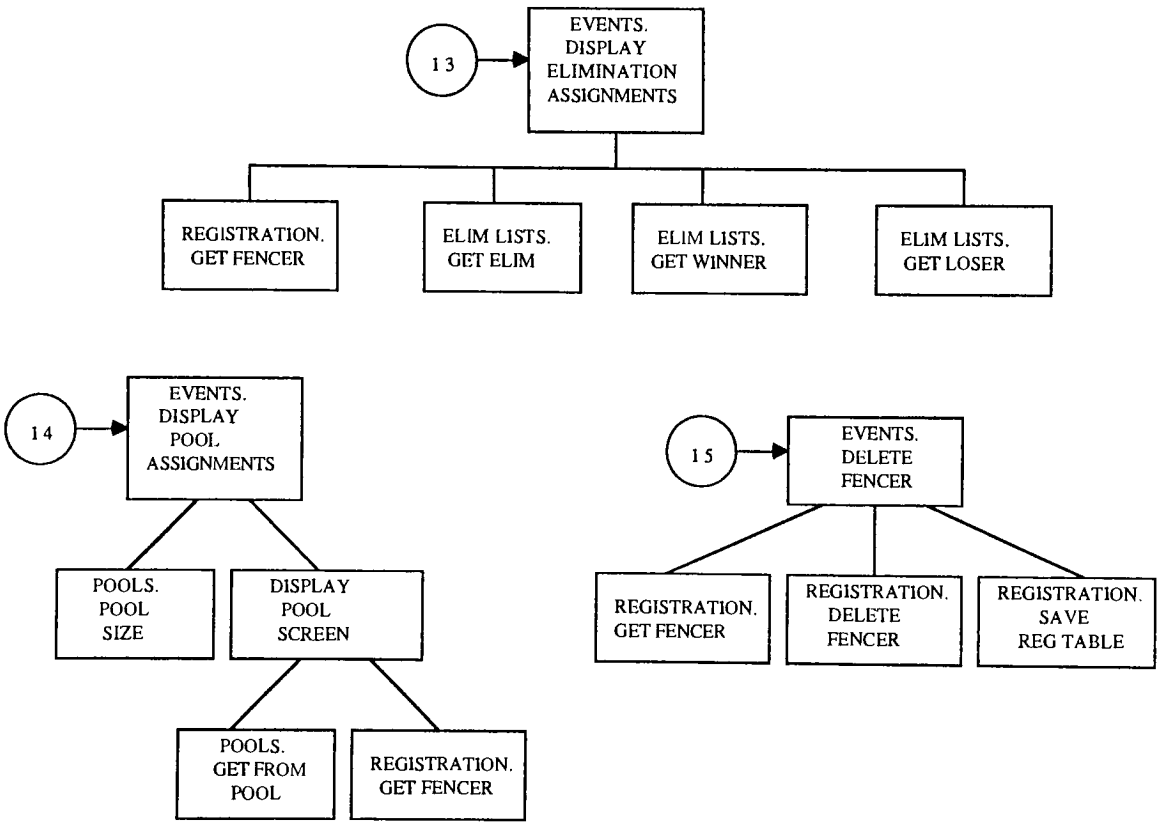

7
SCORE SHEETS.
PRINT
ELIMINATION
FORMS
COMPETITION.
GET
REPORT DATA
ELIM LISTS.
GET
ELIM
ELIM LISTS.
GET
WINNER
ELIM LISTS.
GET
LOSER
REGISTRATION.
GET
FENCER
EVENTS.
GET
VALUES

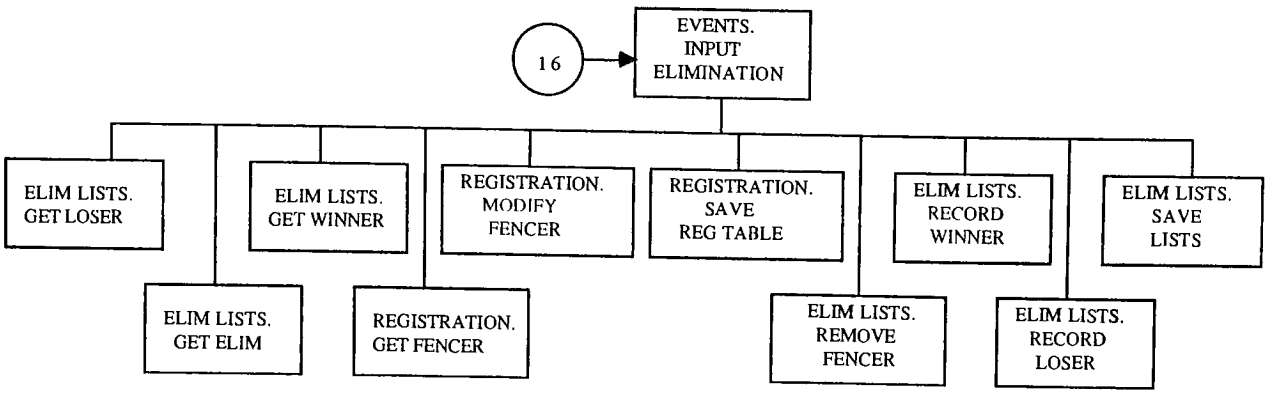

8
SCORE SHEETS.
PRINT
POOL FORMS
COMPETITION.
GET
REPORT DATA
POOLS.
POOL SIZE
POOLS.
GET
FROM POOL
REGISTRATION.
GET
FENCER
EVENTS.
GET
VALUES

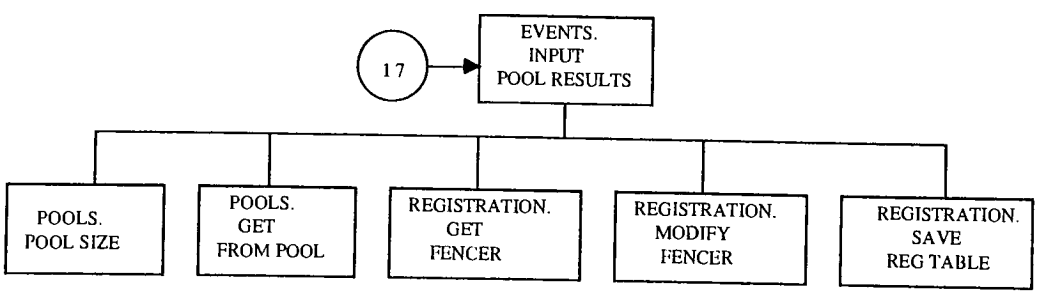

9
REPORTS.
SPECIFY
REPORTS
REPORT
BY
ID
REPORT
BY
RANK
REPORT
BY
NAME
REGISTRATION.
GET
FENCER
REGISTRATION.
READ
BY RANK
REGISTRATION.
READ
BY NAME
COMPETITION.
GET
REPORT DATA
EVENTS.
GET
VALUES

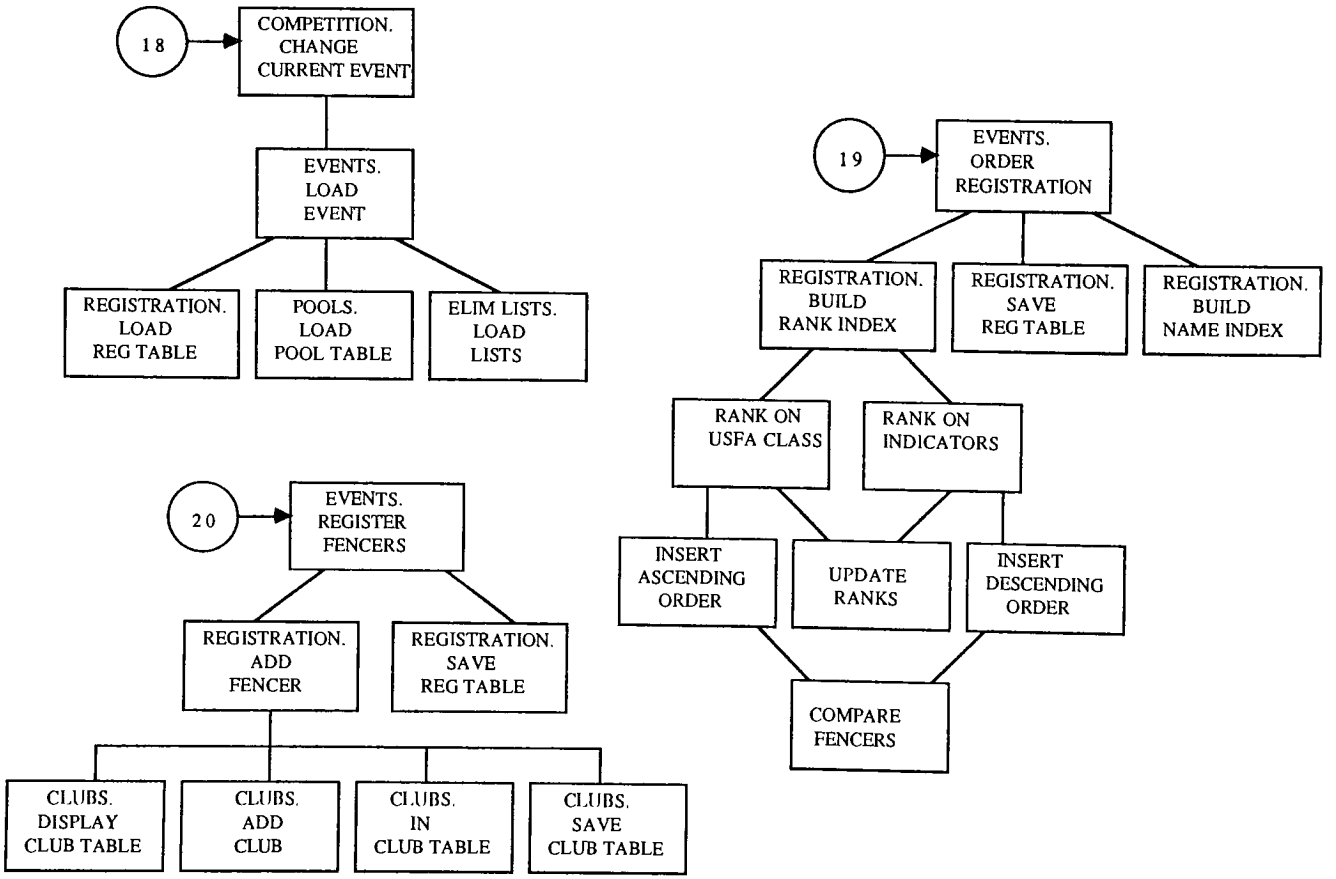

10
EVENTS.
ASSIGN
ELIMINATION
REGISTRATION.
READ
BY RANK
ELIM LISTS.
INSERT
FENCER
ELIM LISTS.
SAVE
LISTS

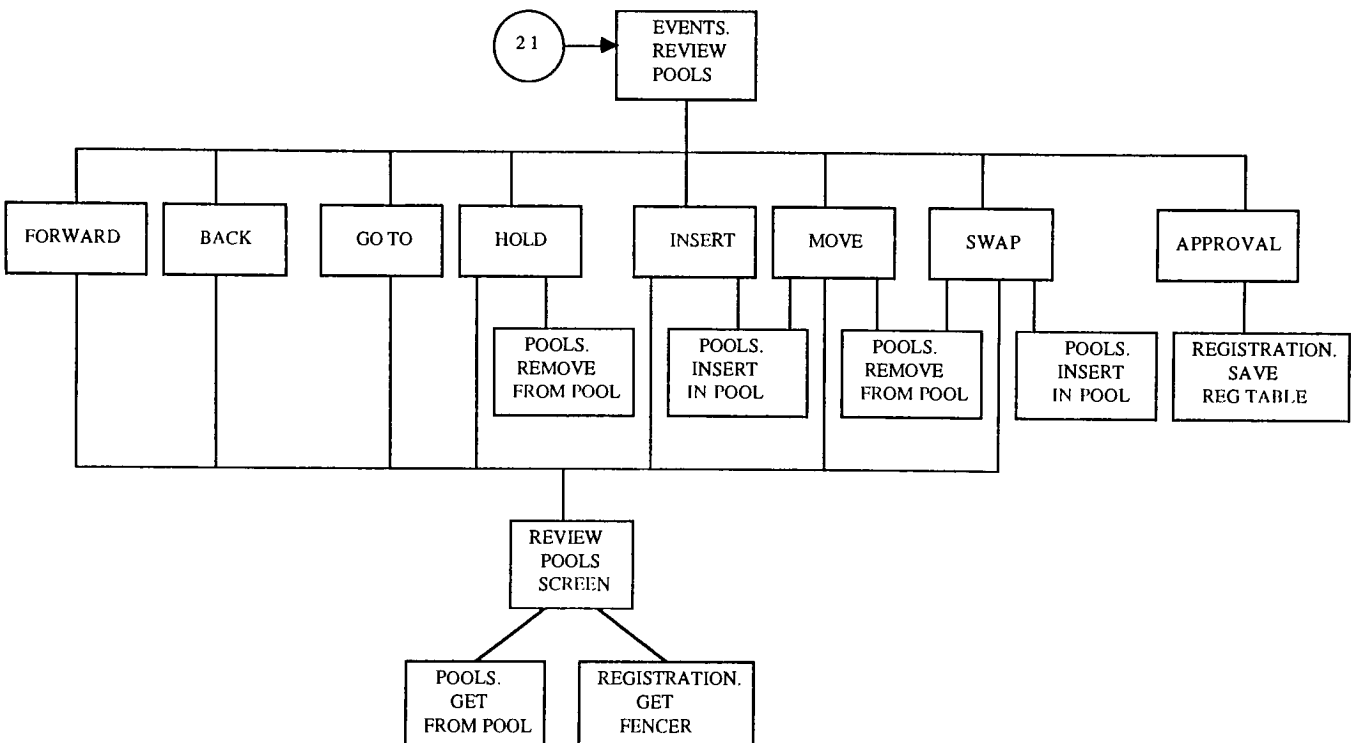

11
EVENTS.
ASSIGN POOLS
GET POOL
PARAMETERS
REGISTRATION.
READ
BY RANK
LOAD
POOLS
POOLS.
INSERT
IN POOL
REARRANGE
POOLS
POOLS.
SAVE
POOL TABLE
POOLS.
POOL SIZE
POOLS.
GET FROM
POOLS
REGISTRATION.
GET
FENCER
POOLS.
REMOVE
FROM POOL
POOLS.
INSERT
IN POOL
12
EVENTS.
CORRECT
REGISTRATION
REGISTRATION.
GET FENCER
CORRECTION
SCREEN
REGISTRATION.
MODIFY
FENCER
REGISTRATION.
SAVE
REG TABLE


13
EVENTS.
DISPLAY
ELIMINATION
ASSIGNMENTS
REGISTRATION.
GET FENCER
ELIM LISTS.
GET ELIM
ELIM LISTS.
GET WINNER
ELIM LISTS.
GET LOSER
14
EVENTS.
DISPLAY
POOL
ASSIGNMENTS
POOLS.
POOL
SIZE
DISPLAY
POOL
SCREEN
POOLS.
GET FROM
POOL
REGISTRATION.
GET FENCER
15
EVENTS.
DELETE
FENCER
REGISTRATION.
GET FENCER
REGISTRATION.
DELETE
FENCER
REGISTRATION.
SAVE
REG TABLE


16
EVENTS.
INPUT
ELIMINATION
ELIM LISTS.
GET LOSER
ELIM LISTS.
GET WINNER
REGISTRATION.
MODIFY
FENCER
REGISTRATION.
SAVE
REG TABLE
ELIM LISTS.
RECORD
WINNER
ELIM LISTS.
SAVE
LISTS
ELIM LISTS.
GET ELIM
REGISTRATION.
GET FENCER
ELIM LISTS.
REMOVE
FENCER
ELIM LISTS.
RECORD
LOSER


17
EVENTS.
INPUT
POOL RESULTS
POOLS.
POOL SIZE
POOLS.
GET
FROM POOL
REGISTRATION.
GET
FENCER
REGISTRATION.
MODIFY
FENCER
REGISTRATION.
SAVE
REG TABLE


18
COMPETITION.
CHANGE
CURRENT EVENT
EVENTS.
LOAD
EVENT
REGISTRATION.
LOAD
REG TABLE
POOLS.
LOAD
POOL TABLE
ELIM LISTS.
LOAD
LISTS
19
EVENTS.
ORDER
REGISTRATION
REGISTRATION.
BUILD
RANK INDEX
REGISTRATION.
SAVE
REG TABLE
REGISTRATION.
BUILD
NAME INDEX
RANK ON
USFA CLASS
RANK ON
INDICATORS
INSERT
ASCENDING
ORDER
UPDATE
RANKS
INSERT
DESCENDING
ORDER
COMPARE
FENCERS
20
EVENTS.
REGISTER
FENCERS
REGISTRATION.
ADD
FENCER
REGISTRATION.
SAVE
REG TABLE
CLUBS.
DISPLAY
CLUB TABLE
CLUBS.
ADD
CLUB
CLUBS.
IN
CLUB TABLE
CLUBS.
SAVE
CLUB TABLE


21
EVENTS.
REVIEW
POOLS
FORWARD
BACK
GO TO
HOLD
INSERT
MOVE
SWAP
APPROVAL
POOLS.
REMOVE
FROM POOL
POOLS.
INSERT
IN POOL
POOLS.
REMOVE
FROM POOL
POOLS.
INSERT
IN POOL
REGISTRATION.
SAVE
REG TABLE
REVIEW
POOLS
SCREEN
POOLS.
GET
FROM POOL
REGISTRATION.
GET
FENCER

APPENDIX H

ADA DESIGN FOR HIGH LEVEL MODULES

```
--                    ****************************
--                    *                          *
--                    *      * READY FENCE!      *
--                    *                          *
--                    ****************************

with TEXT_IO,
     OPERATORS,
     CLUBS,
     COMPETITION,
     REPORTS,
     SCORE_SHEETS;
use TEXT_IO;

procedure READY_FENCE is

     type ACTION_TYPE is (T, C);
          -- T = permit table actions only
          -- C = manage competition and update tables

     ACCEPTED             :  BOOLEAN;
     VALID                :  BOOLEAN;
     ACTION_CLASS         :  ACTION_TYPE;
     INPUT_ACTION         :  CHARACTER;
     COMPETITION_TITLE    :  COMPETITION.C_TITLE_TYPE;
     INPUT_TITLE          :  STRING (1..40);
     PERMISSION           :  OPERATORS.PERMISSION_TYPE;
     RESTART              :  BOOLEAN := TRUE;

     procedure DISPLAY_ENTRY_SCREEN is separate;

     procedure DISPLAY_SIGNON_SCREEN
               (ACCEPTED:  out BOOLEAN;
               PERMISSION: out OPERATORS.PERMISSION_TYPE)
                    is separate;

     procedure DISPLAY_EXIT_SCREEN (RESTART: out BOOLEAN)
                    is separate;

     procedure PERFORM_MENU_FUNCTIONS
               (ACTION_CLASS: in ACTION_TYPE;
               PERMISSION: in OPERATORS.PERMISSION_TYPE)
                    is separate;
```

```
begin     -- READY_FENCE --

     DISPLAY_ENTRY_SCREEN;
     DISPLAY_SIGNON_SCREEN (ACCEPTED, PERMISSION);
     COMPETITION.LOAD_COMPETITION;
     CLUBS.LOAD_CLUB_TABLE;
     OPERATORS.LOAD_SECURITY_TABLE;

     while (RESTART) loop
          if COMPETITION.IS_OPEN then
               ACTION_CLASS := C;  -- manage competition
          else
               VALID := FALSE;
               while (not VALID) loop
                   put
          ("ENTER C TO OPEN COMPETITION OR T TO UPDATE
                                                    TABLES");
                   get (INPUT_ACTION);
                   if INPUT_ACTION = 'C'
                      or
                      INPUT_ACTION = 'T' then
                          VALID := TRUE;
                          if INPUT_ACTION = 'C' then
                              ACTION_CLASS := C;
                          else
                              ACTION_CLASS := T;
                          end if;
                   else
                          put_line (" INVALID - TRY AGAIN");

                   end if;
               end loop;
               if ACTION_CLASS = C then
                    put ("ENTER TITLE OF COMPETITION: ");
                    get (INPUT_TITLE);

                    COMPETITION_TITLE :=
                     COMPETITION.C_TITLE_TYPE(INPUT_TITLE);

                    COMPETITION.OPEN_COMPETITION
                                       (COMPETITION_TITLE);
               end if;
          end if;
          COMPETITION.INITIALIZE_CURRENT_EVENT;
          PERFORM_MENU_FUNCTIONS (ACTION_CLASS, PERMISSION);
          DISPLAY_EXIT_SCREEN (RESTART);
     end loop;

end READY_FENCE;
```

```
--              ************************************
--              *                                  *
--              *           READY_FENCE.           *
--              *      PERFORM_MENU_FUNCTIONS      *
--              *                                  *
--              ************************************

with CLUBS, CONSTANTS, COMPETITION,
     OPERATORS, REPORTS, SCORE_SHEETS;

separate (READY_FENCE)
procedure PERFORM_MENU_FUNCTIONS
          (ACTION_CLASS: in ACTION_TYPE;
           PERMISSION: in OPERATORS.PERMISSION_TYPE) is
          --  ACTION_CLASS:
          --       T = permit table actions only
          --       C = manage competition and update tables

     type STRING2_TYPE is new STRING (1..2);
     type MENU_FUNCTION_TYPE is ( BLANK, INVALID, SECURITY,
                                  E,  M, AE, AP, CR, DC, DE,
                                  DP, DR, IE, IP, O, PE, PP,
                                  PR, R,  RP, TE, UC, W);

     ACCEPTED                 :  BOOLEAN;
     CLUBS_NEXT_ACTION        :  CLUBS.ACTION_TYPE;
     COMPETITION_NEXT_ACTION  :  COMPETITION.ACTION_TYPE;
     NEXT_ACTION              :  MENU_FUNCTION_TYPE  := M;
     OPERATORS_NEXT_ACTION    :  OPERATORS.ACTION_TYPE;
     STRING_ACTION            :  STRING2_TYPE;

     procedure CHECK_PERMISSION
               (ACTION     : in MENU_FUNCTION_TYPE;
               PERMISSION : in OPERATORS.PERMISSION_TYPE;
               ACCEPTED   : out BOOLEAN)  is separate;

     procedure CONVERT_MENU_FUNCTION
               (STRING2       : in  STRING2_TYPE;
                MENU_FUNCTION : out MENU_FUNCTION_TYPE)
                is separate;
               -- Convert string of length 2 to
               --                          MENU_FUNCTION_TYPE
               -- If blank, value = BLANK
               -- If = CONSTANTS.SECURITY_ACTION,
               --                          value = SECURITY
               -- If not a valid function, value = INVALID

    procedure DISPLAY_MENU (NEXT_ACTION : out STRING2_TYPE)
               is separate;
               -- input and return next action without
               --     validation
```

```
begin    -- PERFORM_MENU_FUNCTIONS --

     while NEXT_ACTION /= E  loop
          if NEXT_ACTION = M  or NEXT_ACTION = BLANK then
               DISPLAY_MENU (STRING_ACTION);
               CONVERT_MENU_FUNCTION
                         (STRING_ACTION, NEXT_ACTION);
          else
               CHECK_PERMISSION
                    (NEXT_ACTION, PERMISSION, ACCEPTED);
               if not ACCEPTED then
                    put
                    ("ACTION NOT PERMITTED FOR OPERATOR");
                    DISPLAY_MENU (STRING_ACTION);
                    CONVERT_MENU_FUNCTION
                         (STRING_ACTION, NEXT_ACTION);
               else
                    if ACTION_CLASS = T then
                         case NEXT_ACTION is

                              when SECURITY =>
                                   OPERATORS.SECURITY_MENU
                                    (OPERATORS_NEXT_ACTION);
                                  STRING_ACTION :=
          STRING2_TYPE (STRING (OPERATORS_NEXT_ACTION));
                                   CONVERT_MENU_FUNCTION
                                        (STRING_ACTION,
                                         NEXT_ACTION);
                              when UC =>
                                   CLUBS.CLUB_MENU
                                        (CLUBS_NEXT_ACTION);

                                   STRING_ACTION :=
            STRING2_TYPE (STRING(CLUBS_NEXT_ACTION));
                                   CONVERT_MENU_FUNCTION
                                        (STRING_ACTION,
                                         NEXT_ACTION);
                              when DC =>
                                    -- display competition
                                  COMPETITION.PERFORM_ACTION
                                  (COMPETITION_NEXT_ACTION);
                                  STRING_ACTION :=
         STRING2_TYPE(STRING(COMPETITION_NEXT_ACTION));
                                  CONVERT_MENU_FUNCTION
                                        (STRING_ACTION,
                                        NEXT_ACTION);
```

```
                                when others =>
                                     put
                              ("ONLY TABLE ACTIONS PERMITTTED");
                                     DISPLAY_MENU
                                          (STRING_ACTION);
                                      CONVERT_MENU_FUNCTION
                                          (STRING_ACTION,
                                          NEXT_ACTION);
                           end case;
                      else  -- ACTION_CLASS = C
                           case NEXT_ACTION is

                                when AE | AP | CR | DC |
                                     DE | DP | DR | O  |
                                     R  | RP | TE | W     =>

                                COMPETITION.PERFORM_ACTION
                                    (COMPETITION_NEXT_ACTION);
                                 STRING_ACTION :=
      STRING2_TYPE (STRING (COMPETITION_NEXT_ACTION));
                                 CONVERT_MENU_FUNCTION
                                     (STRING_ACTION,
                                     NEXT_ACTION);

                                when PE =>
                        SCORE_SHEETS.PRINT_ELIMINATION_FORMS;

                                when PP =>
                               SCORE_SHEETS.PRINT_POOL_FORMS;

                                when PR =>
                                     REPORTS.SPECIFY_REPORT;

                                when others =>
                                     put ("INVALID CHOICE");
                                     DISPLAY_MENU
                                          (STRING_ACTION);
                                     CONVERT_MENU_FUNCTION
                                          (STRING_ACTION,
                                          NEXT_ACTION);

                           end case;
                      end if;
                 end if;
            end if;
       end loop;

end PERFORM_MENU_FUNCTIONS;
```

```
--         ********************************************
--         *                                          *
--         *    READY_FENCE.PERFORM_MENU_FUNCTIONS    *
--         *              CHECK_PERMISSION            *
--         *                                          *
--         ********************************************

with OPERATORS; use OPERATORS;
separate (READY_FENCE.PERFORM_MENU_FUNCTIONS)
procedure CHECK_PERMISSION
    (ACTION     : in MENU_FUNCTION_TYPE;
     PERMISSION : in OPERATORS.PERMISSION_TYPE;
     ACCEPTED   : out BOOLEAN) is

begin

     ACCEPTED := TRUE;
     case ACTION  is
          when SECURITY  =>
               if PERMISSION /= S then
                    ACCEPTED := FALSE;
               end if;
          when AE | AP | CR | IE | IP | O  |
               R  | RP | UC | W  =>  --  update functions
               if PERMISSION = R then
                    ACCEPTED := FALSE;
               end if;
          when others =>
               ACCEPTED := FALSE;
     end case;

end CHECK_PERMISSION;
```

```
--                  ****************************
--                  *                          *
--                  *        * CLUBS           *
--                  *                          *
--                  ****************************

package CLUBS is

     type CLUB_CODE_TYPE is new STRING (1..4);
     type CLUB_NAME_TYPE is new STRING (1..20);
     type ACTION_TYPE is new STRING (1..2);

     type CLUB_TYPE is
       record
          CLUB_CODE           :  CLUB_CODE_TYPE;
          AFFINITY_GROUP      :  CLUB_CODE_TYPE;
          CLUB_NAME           :  CLUB_NAME_TYPE;
       end record;

     function IN_CLUB_TABLE (CLUB_CODE : in CLUB_CODE_TYPE)
                            return BOOLEAN;

     procedure ADD_CLUB (CLUB    : in CLUB_TYPE;
                         ACCEPTED: out BOOLEAN);

     procedure CLUB_MENU (NEXT_ACTION : out ACTION_TYPE);
          --   Save club table after CONSTANT.SAVE_POINT
          --        updates

     procedure DISPLAY_CLUB_TABLE
               (FIRST_LETTER : in CHARACTER);

     procedure LOAD_CLUB_TABLE;
          --   read club table file and load into array

          --   count number of clubs in the table
          --   also set up pointer arrays for name list and
          --       code list

     procedure SAVE_CLUB_TABLE;

end CLUBS;
```

```
with CONSTANTS, TEXT_IO;
use TEXT_IO;

package body CLUBS is

     type CLUB_INDEX_TYPE is range 1 .. CONSTANTS.MAX_CLUBS;

     type CLUB_REC_TYPE is
       record
          CLUB                  :  CLUB_TYPE;
          CODE_INDEX            :  CLUB_INDEX_TYPE;
               --  subscript in sorted code array
          NAME_INDEX            :  CLUB_INDEX_TYPE;
               --  subscript in sorted name array
       end record;

     type CLUB_TABLE_TYPE is array
                         (1 .. CLUB_INDEX_TYPE'LAST)
                       of CLUB_REC_TYPE;

     type CLUB_POINTER_ARRAY_TYPE is array
                         (1 .. CLUB_INDEX_TYPE'LAST)
                         of CLUB_INDEX_TYPE;

               ----------------------------------------
               --                                    --
               --        CLUBS GLOBAL DATA           --
               --                                    --
               ----------------------------------------

     CLUB_TABLE : CLUB_TABLE_TYPE;
     NAME_ARRAY : CLUB_POINTER_ARRAY_TYPE;  -- name index
     CODE_ARRAY : CLUB_POINTER_ARRAY_TYPE;  -- code index
     N_CLUBS    : CLUB_INDEX_TYPE;     -- number of clubs
                                       --         in table

               ----------------------------------------

     function IN_CLUB_TABLE
               (CLUB_CODE : in CLUB_CODE_TYPE)
               return BOOLEAN is separate;

     procedure ADD_CLUBS is separate;
          --  present add clubs screen to get club data
          --  call ADD_CLUB to add to club table

     procedure DELETE_CLUB (CLUB_CODE : in CLUB_CODE_TYPE)
               is separate;

     procedure DISPLAY_CLUB_TABLE
               (FIRST_LETTER : in CHARACTER) is separate;
```

```
    procedure INPUT_CLUB is separate;
         --   Input club data
         --   Add club record to club table

    procedure LOAD_CLUB_TABLE is separate;
         --   Read club table file and load into array
         --   Count number of clubs in the table
         --   Set up pointer arrays for name list and
         --        code list

    procedure MODIFY_CLUB (CLUB : in CLUB_CODE_TYPE)
              is separate;

    procedure PRINT_CLUB_TABLE is separate;

    procedure SAVE_CLUB_TABLE is separate;

    procedure SEARCH_CLUB_CODES
              (CLUB_CODE : in CLUB_CODE_TYPE;
               FOUND     : out BOOLEAN;
               LOCATION  : out CLUB_INDEX_TYPE)
               is separate;
         --   Find club in table using club code

    procedure SEARCH_CLUB_NAMES
              (CLUB_NAME : in CLUB_NAME_TYPE;
               FOUND     : out BOOLEAN;
               LOCATION  : out CLUB_INDEX_TYPE)
               is separate;
         --   Find club in table using club name

    procedure ADD_CLUB
              (CLUB     : in CLUB_TYPE;
              ACCEPTED : out BOOLEAN) is separate;

    procedure CLUB_MENU (NEXT_ACTION : out ACTION_TYPE)
              is separate;

end CLUBS;
```

```
--              ************************************
--              *                                  *
--              *          CLUBS.ADD_CLUB          *
--              *                                  *
--              ************************************

with CONSTANTS, TEXT_IO;
use TEXT_IO;
separate (CLUBS)

procedure ADD_CLUB
          (CLUB     : in CLUB_TYPE;
           ACCEPTED : out BOOLEAN) is

     FOUND          :    BOOLEAN;
     CODE_LOCATION  :    CLUB_INDEX_TYPE;
     ID             :    CLUB_INDEX_TYPE;
     NAME_LOCATION  :    CLUB_INDEX_TYPE;
     NEW_N_CLUBS    :    CLUB_INDEX_TYPE;
     IO_CLUB_NAME   :  STRING (1..20);
     IO_CLUB_CODE   :  STRING (1..4) ;

     procedure INSERT_CLUB_NAME
     (CLUB_NAME     : in CLUB_NAME_TYPE;
      NAME_LOCATION : in CLUB_INDEX_TYPE) is separate;
         --   move items in NAME_ARRAY down
         --   insert N_CLUBS at NAME_LOCATION + 1
         --   add 1 to NAME_INDEX in CLUB_TABLE for
         --        all items that were moved down in
         --        NAME_ARRAY

     procedure INSERT_CLUB_CODE
      (CLUB_CODE     : in CLUB_CODE_TYPE;
       CODE_LOCATION : in CLUB_INDEX_TYPE) is separate;
         --   move items in CODE_ARRAY down
         --   insert N_CLUBS at CODE_LOCATION + 1
         --   add 1 to CODE_INDEX in CLUB_TABLE for
         --         all items that were moved down in
         --         CODE_ARRAY
```

```
begin    --   ADD_CLUB   --

     SEARCH_CLUB_CODES
          (CLUB.CLUB_CODE, FOUND, CODE_LOCATION);
     if FOUND then
          ACCEPTED := FALSE;
          ID := CODE_ARRAY(CODE_LOCATION);
          IO_CLUB_NAME :=
               STRING(CLUB_TABLE(ID).CLUB.CLUB_NAME);
          PUT ( "ALREADY IN TABLE. NAME IS " );
          PUT_LINE (IO_CLUB_NAME);
     else
          SEARCH_CLUB_NAMES
               (CLUB.CLUB_NAME, FOUND, NAME_LOCATION);
          if FOUND then
               ACCEPTED := FALSE;
               ID := NAME_ARRAY(NAME_LOCATION);
               IO_CLUB_CODE :=
                    STRING(CLUB_TABLE(ID).CLUB.CLUB_CODE);
               PUT ( "ALREADY IN TABLE. CODE IS " );
               PUT_LINE ( IO_CLUB_CODE );
          else      --   name and code are not in table
               NEW_N_CLUBS := N_CLUBS + 1;
               if NEW_N_CLUBS > CONSTANTS.MAX_CLUBS then
                    ACCEPTED := FALSE;
               else  --  club fits in table
                    ACCEPTED := TRUE;
                    INSERT_CLUB_NAME
                              (CLUB.CLUB_NAME,
                               NAME_LOCATION);
                    INSERT_CLUB_CODE
                              (CLUB.CLUB_CODE,
                               CODE_LOCATION);
                    N_CLUBS := NEW_N_CLUBS;
                    CLUB_TABLE(N_CLUBS).CLUB := CLUB;
                    CLUB_TABLE(N_CLUBS).CODE_INDEX :=
                                        CODE_LOCATION + 1;
                    CLUB_TABLE(N_CLUBS).NAME_INDEX :=
                                        NAME_LOCATION + 1;
               end if;
          end if;
     end if;

end ADD_CLUB;
```

```
--                      ****************************
--                      *                          *
--                      *     CLUBS.CLUB_MENU      *
--                      *                          *
--                      ****************************

with CONSTANTS, TEXT_IO;
use TEXT_IO;

separate (CLUBS)
procedure CLUB_MENU (NEXT_ACTION : out ACTION_TYPE) is

     CLUB_CODE        :  CLUB_CODE_TYPE;
     IO_CLUB_CODE     :  STRING (1 .. 4);
     IO_NEXT_ACTION   :  STRING (1 .. 2);
     TABLE_UPDATED    :  BOOLEAN;
     UPDATE_ACTION    :  CHARACTER   := ' ';
     UPDATE_COUNT     :  NATURAL;
     VALID            :  BOOLEAN;

     procedure DISPLAY_CLUB_MENU is separate;
```

```
begin    -- CLUB_MENU --

     while UPDATE_ACTION /= 'E' loop
          TABLE_UPDATED := FALSE;    -- initialize
          UPDATE_COUNT := 0;         --  determines when
                                     --   to save table
          DISPLAY_CLUB_MENU;
          get (UPDATE_ACTION);
          VALID := FALSE;
          while not VALID loop
               case UPDATE_ACTION is
                    when 'A' | 'C' | 'D' |
                         'L' | 'P' | 'E' =>
                              VALID := TRUE;
                    when others =>
                         put ("INVALID CHOICE; REENTER: ");
               end case;
          end loop;
          if UPDATE_ACTION = 'C' or UPDATE_ACTION = 'D' then
               put ("ENTER CLUB CODE: ");
               get (IO_CLUB_CODE);
               CLUB_CODE := CLUB_CODE_TYPE (IO_CLUB_CODE);
               TABLE_UPDATED := TRUE;
          end if;
          case UPDATE_ACTION is
               when 'A' => ADD_CLUBS;
               when 'C' => MODIFY_CLUB (CLUB_CODE);
               when 'D' => DELETE_CLUB (CLUB_CODE);
               when 'L' => DISPLAY_CLUB_TABLE (' ');
               when 'P' => PRINT_CLUB_TABLE;
               when others => null;
          end case;
          if TABLE_UPDATED then
               UPDATE_COUNT := UPDATE_COUNT + 1;
          end if;
          if UPDATE_COUNT = CONSTANTS.CLUB_SAVE_POINT then
               SAVE_CLUB_TABLE;
               UPDATE_COUNT := 0;
          end if;
     end loop;

     if UPDATE_COUNT > 0 then
          SAVE_CLUB_TABLE;
     end if;
     get (IO_NEXT_ACTION);
     NEXT_ACTION := ACTION_TYPE (IO_NEXT_ACTION);


end CLUB_MENU;
```

```
--                    ****************************
--                    *                          *
--                    *      * COMPETITION       *
--                    *                          *
--                    ****************************

with CONSTANTS,
     EVENTS;

package COMPETITION is

     type C_STATUS_TYPE is private;

     type C_TITLE_TYPE is new STRING (1 .. 40);

     type ACTION_TYPE is new string (1 .. 2);
         -- values:
         -- AE =  assign elimination
         -- A  = assign pools
         -- CR = correct  registration
         -- DC = display competition status
         -- DE = display elimination assignments
         -- DP = display pool assignments
         -- DR = delete a registered fencer
         -- IE = input elimination results
         -- IP = input pool results
         -- O  = order list of fencers
         -- R  = register fencers
         -- RP = review pools
         -- TE = transfer to different current event
         -- W  = withdraw a fencer (between rounds)

     type REPORT_DATA_TYPE is
       record
          RPT_C_TITLE          :  C_TITLE_TYPE;    -- centered
          RPT_E_TITLE          :  EVENTS.CTITLE_TYPE;
                                                   -- centered
          ROUNDS_POOLS         :  NATURAL;
          ROUNDS_ELIM          :  NATURAL;
          N_POOLS              :  NATURAL;
          N_REGISTERED         :  NATURAL;
          N_COMPETING          :  NATURAL;
          TYPE_OF_ELIM         :  CHARACTER;
          RANK_INDEX_ROOT      :  NATURAL;
          NAME_INDEX_ROOT      :  NATURAL;
          RANK_INDICATOR       :  BOOLEAN;
                                     -- is sorted on rank
          NAME_INDICATOR       :  BOOLEAN;
                                      -- is sorted on name
       end record;

     function IS_OPEN return BOOLEAN;
```

```
     procedure CLOSE_COMPETITION;

     procedure CLOSE_CURRENT_EVENT;

     procedure GET_REPORT_DATA
               (REPORT_DATA : out REPORT_DATA_TYPE);

     procedure INITIALIZE_CURRENT_EVENT;
          --   display the competition in short form
          --   operator may choose existing open event
          --        or may open a new event
          --   if no available new event, prompt for event
          --         to close

     procedure LOAD_COMPETITION;

     procedure OPEN_COMPETITION (C_TITLE : C_TITLE_TYPE);
          --   initialize all event records

     procedure PERFORM_ACTION
               (ACTION : in out ACTION_TYPE);
          --   apply function to current event
          --   return operator input for next action

     procedure SAVE_COMPETITION;

private

     type C_STATUS_TYPE is (O,C);

end COMPETITION;
```

```
with CONSTANTS,
     EVENTS;
package body COMPETITION is

               ------------------------------------------
               --                                      --
               --       COMPETITION GLOBAL DATA        --
               --                                      --
               ------------------------------------------

          C_STATUS             :  C_STATUS_TYPE;
          C_TITLE              :  C_TITLE_TYPE;   -- centered
          CURRENT_EVENT_NO     :  EVENTS.EVENT_NUMBER_TYPE;
          EVENT_TABLE          :  array
                                    (1 .. CONSTANTS.N_EVENTS)
                                    of EVENTS.EVENT_TYPE;

               ------------------------------------------
     function IS_OPEN return BOOLEAN is separate;

     procedure CHANGE_CURRENT_EVENT is separate;
          --   prompt for new event number
          --   then load selected event into memory from
          --    storage

     procedure CLOSE_COMPETITION is separate;

     procedure CLOSE_CURRENT_EVENT is separate;

     procedure DISPLAY_COMPETITION is separate;
          --   display the competition in detail using the
          --        event records

     procedure GET_REPORT_DATA
               (REPORT_DATA : out REPORT_DATA_TYPE)
               is separate;

     procedure INITIALIZE_CURRENT_EVENT is separate;
          --   display the competition in short form
          --   operator may choose existing open event
          --        or may open a new event:
          --        EVENT.OPEN_EVENT
          --   if no available new event, prompt for event
          --        to close and then EVENT.CLOSE_EVENT
          --   if existing open event is chosen,
          --        EVENT.LOAD_EVENT

     procedure LOAD_COMPETITION is separate;
```

```
     procedure OPEN_COMPETITION (C_TITLE : C_TITLE_TYPE)
               is separate;
          --   initialize all event records

     procedure PERFORM_ACTION
               (ACTION : in out ACTION_TYPE)
               is separate;
          --   apply function to current event
          --   return operator input for next action

     procedure SAVE_COMPETITION is separate;

end COMPETITION;
```

```
--                    ****************************
--                    *                          *
--                    *       * CONSTANTS        *
--                    *                          *
--                    ****************************


package CONSTANTS is

     CLUB_SAVE_POINT           : constant         :=     15;
     MAX_CLUBS                 : constant         :=    100;
     MAX_FENCERS               : constant         :=    175;
     MAX_IN_POOL               : constant         :=      7;
     MAX_LINES                 : constant         :=     55;
     MAX_OPERATORS             : constant         :=     15;
     MAX_POINTS                : constant         :=     18;
     MAX_POOLS                 : constant         :=     25;
     N_EVENTS                  : constant         :=      4;
     REGISTRATION_SAVE_POINT   : constant         :=     10;
     SECURITY_ACTION           : constant string  :=   "AL";
     UNATTACHED_INDICATOR      : constant string  := "UNAT";

end CONSTANTS;
```

```
--                    ****************************
--                    *                          *
--                    *      * ELIM_LISTS        *
--                    *                          *
--                    ****************************

with CONSTANTS,
     FENCERS;

package ELIM_LISTS is

     procedure GET_ELIM (INDEX  : in  NATURAL;
                         FENCER : out FENCERS.FID_TYPE);

     procedure GET_LOSER (INDEX  : in  NATURAL;
                          FENCER : out FENCERS.FID_TYPE);

     procedure GET_WINNER (INDEX  : in  NATURAL;
                           FENCER : out FENCERS.FID_TYPE);

     procedure INITIALIZE;

     procedure LOAD_LISTS;

     procedure INSERT_FENCER
               (ID       :  in FENCERS.FID_TYPE;
                POSITION :  in INTEGER);
          --   Put fencer into elimination list

     procedure RECORD_LOSER (ID : in FENCERS.FID_TYPE);
          --   Put fencer into list of losers

     procedure RECORD_WINNER (ID : in FENCERS.FID_TYPE);
          --   Put fencer into list of winners

     procedure REMOVE_FENCER (ID : in FENCERS.FID_TYPE);

     procedure SAVE_LISTS;

end ELIM_LISTS;
```

```
with FENCERS;
package body ELIM_LISTS is

                -----------------------------------------
                --                                     --
                --        ELIM_LISTS GLOBAL DATA       --
                --                                     --
                -----------------------------------------

     ELIM_LIST      :  array (1 .. 32) of FENCERS.FID_TYPE;
     LOSER_LIST     :  array (1 .. 16) of FENCERS.FID_TYPE;
     WINNER_LIST    :  array (1 .. 16) of FENCERS.FID_TYPE;

                -----------------------------------------

     procedure GET_ELIM
               (INDEX  : in  NATURAL;
               FENCER : out FENCERS.FID_TYPE) is separate;

     procedure GET_LOSER
               (INDEX  : in  NATURAL;
               FENCER : out FENCERS.FID_TYPE)
               is separate;

     procedure GET_WINNER
               (INDEX  : in  NATURAL;
               FENCER : out FENCERS.FID_TYPE)
               is separate;

     procedure LOAD_LISTS is separate;

     procedure INITIALIZE is separate;

     procedure INSERT_FENCER
               (ID       :  in FENCERS.FID_TYPE;
               POSITION :  in INTEGER) is separate;
          --   Put fencer into elimination list

     procedure RECORD_LOSER (ID : in FENCERS.FID_TYPE)
               is separate;
          --   Put fencer into list of losers

     procedure RECORD_WINNER (ID : in FENCERS.FID_TYPE)
               is separate;
          --   Put fencer into list of winners

     procedure REMOVE_FENCER (ID : in FENCERS.FID_TYPE)
               is separate;

     procedure SAVE_LISTS is separate;

end ELIM_LISTS;
```

```
--                    ****************************
--                    *                          *
--                    *        * EVENTS          *
--                    *                          *
--                    ****************************

with CONSTANTS,
     ELIM_LISTS,          -- elimination lists
     FENCERS,
     POOLS,
     REGISTRATION;

package EVENTS is

     type EVENT_TYPE is limited private;

     type CTITLE_TYPE is new STRING (1 .. 16);
          -- centered
     type ELIM_TYPE is (D,R);
          -- D = direct   R = with repechage
     type EVENT_NUMBER_TYPE is range
                              1 .. CONSTANTS.N_EVENTS;
     type INPOOL_TYPE is range 0 .. CONSTANTS.MAX_IN_POOL;
     type LTITLE_TYPE is new STRING (1 .. 16);
          -- left justified
     type NPOOL_TYPE is range 0 .. CONSTANTS.MAX_POOLS;
     type POOL_NUMBER_ARRAY_TYPE is
               array  (1 .. CONSTANTS.MAX_POOLS)
                       of NPOOL_TYPE;
     type STATUS_TYPE is (I,O,C);
          --  I  initial   O = open    C = closed

     type VALUE_TYPE is        -- event for read only
       record
          EVENT_NUMBER         :  EVENT_NUMBER_TYPE;
          EVENT_STATUS         :  STATUS_TYPE;
          TITLE_L              :  LTITLE_TYPE;
                                        -- left justified
          TITLE_C              :  CTITLE_TYPE; -- centered
          N_REGISTERED         :  FENCERS.FID_TYPE;
               -- number enrolled
          N_COMPETING          :  FENCERS.FID_TYPE;
               -- active competitors
          N_POOLS              :  NPOOL_TYPE;
               -- number of pools
          A                    :  NPOOL_TYPE;
               -- number of pools of B fencers
          B                    :  INPOOL_TYPE;
               -- number of fencers in A pools
          C                    :  NPOOL_TYPE;
               -- number of pools of D fencers
```

```
      D                      :  INPOOL_TYPE;
           -- number of fencers in C pools
      RANK_INDEX_ROOT             :  FENCERS.FID_TYPE;
           -- ID of root of rank index
      NAME_INDEX_ROOT             :  FENCERS.FID_TYPE;
           -- ID of root of name index
      N_ROUNDS_POOLS         :  NATURAL;
           -- number of rounds of pools
           -- includes the current round
      N_ROUNDS_ELIM          :  NATURAL;
           -- number of rounds of elimination
           -- includes the current round
      TYPE_OF_ELIM           :  ELIM_TYPE;
           -- direct or with repechage
      N_ELIM_OUTSTANDING     :  FENCERS.FID_TYPE;
           -- number of fencers for whom results have
           -- not been input for round of elimination
      N_POOLS_OUTSTANDING :  NPOOL_TYPE;
           -- number of pools for whom results have not
           -- been input for round of pools
      OUTSTANDING_POOLS      :  POOL_NUMBER_ARRAY_TYPE;
   end record;

procedure ASSIGN_ELIMINATION
           (EVENT : in out EVENT_TYPE);
      --   If first time, get elimination type
      --   Get fencers in order of rank tree
      --   Build elimination list

procedure ASSIGN_POOLS (EVENT : in out EVENT_TYPE);
      --   Get fencers in order of rank tree
      --   Build pool table

procedure CLOSE_EVENT (EVENT : in out  EVENT_TYPE);

procedure CORRECT_REGISTRATION
           (EVENT : in out EVENT_TYPE);
      --   Input fencer data
      --   modify fencer registration record

procedure DELETE_FENCER
           (EVENT : in out EVENT_TYPE);
      --   Input fencer ID
      --   Delete from registration list

procedure DISPLAY_ELIMINATION_ASSIGNMENTS
           (EVENT : in EVENT_TYPE);

procedure DISPLAY_POOL_ASSIGNMENTS
           (EVENT : in EVENT_TYPE);
      --   use pool tables
      --   get fencer data from registration
```

```
 procedure GET_VALUES
           (EVENT  : in EVENT_TYPE;
           VALUES : out VALUE_TYPE);

procedure INPUT_ELIMINATION
           (EVENT : in out EVENT_TYPE);
      --   Input fencer results
      --   Get fencer registration record
      --   Modify fencer registration record

 procedure INPUT_POOL
           (EVENT : in out EVENT_TYPE);
      --   Input pool number and fencer data
      --   Get fencer registration record
      --   Modify fencer registration record

 procedure LOAD_EVENT (EVENT : in out  EVENT_TYPE);
      --   load registration table, pool table, and
      --        elimination  lists

 procedure OPEN_EVENT (EVENT : in out  EVENT_TYPE);

 procedure ORDER_REGISTRATION
           (EVENT : in out EVENT_TYPE);
      --   Input selection for name or rank order
      --   Get fencers in order of ID
      --   Build appropriate index tree

 procedure REGISTER_FENCERS
           (EVENT : in out EVENT_TYPE ;
            NEXT_ACTION : out REGISTRATION.ACTION_TYPE);
      --   Input fencer data
      --   Validate club using club table
      --   Add fencer to registration list

 procedure REVIEW_POOLS
           (EVENT : in EVENT_TYPE);
      --   Display pool assignments
      --   Input commands to change display or move
      --        fencers
      --   Add fencers to pools or remove them

 procedure WITHDRAW_FENCER
           (EVENT : in EVENT_TYPE);
      --   Input fencer ID number
      --   Change fencer's STATUS to W
```

```
private

      type EVENT_TYPE is
        record
           EVENT_NUMBER          :  EVENT_NUMBER_TYPE;
           EVENT_STATUS          :  STATUS_TYPE;
           TITLE_L               :  LTITLE_TYPE;
                -- left justified
           TITLE_C               :  CTITLE_TYPE;
                -- centered
           N_REGISTERED          :  FENCERS.FID_TYPE;
                -- number enrolled
           N_COMPETING           :  FENCERS.FID_TYPE;
                -- active competitors
           N_POOLS               :  NPOOL_TYPE;
                -- number of pools
           A                     :  NPOOL_TYPE;
                -- number of pools of B fencers
           B                     :  INPOOL_TYPE;
                -- number of fencers in A pools
           C                     :  NPOOL_TYPE;
                -- number of pools of D fencers
           D                     :  INPOOL_TYPE;
                -- number of fencers in C pools
           RANK_INDEX_ROOT          :  FENCERS.FID_TYPE;
                -- ID of root of rank index
           NAME_INDEX_ROOT          :  FENCERS.FID_TYPE;
                -- ID of root of name index
           N_ROUNDS_POOLS        :  NATURAL;
                -- number of rounds of pools
                -- includes the current round
           N_ROUNDS_ELIM         :  NATURAL;
                -- number of rounds of elimination
                -- includes the current round
           TYPE_OF_ELIM          :  ELIM_TYPE;
                -- direct or with repechage
           N_ELIM_OUTSTANDING    :  FENCERS.FID_TYPE;
                -- number of fencers for whom results have
                -- not been input for round of elimination
           N_POOLS_OUTSTANDING :  NPOOL_TYPE;
                -- number of pools for whom results have not
                -- been input for round of pools
           OUTSTANDING_POOLS     :  POOL_NUMBER_ARRAY_TYPE;
           RANK_INDICATOR        :  BOOLEAN;
                -- is sorted on rank
           NAME_INDICATOR        :  BOOLEAN;
                -- is sorted on name
        end record;

end EVENTS;
```

```
with CONSTANTS,
     ELIM_LISTS,          -- elimination lists
     FENCERS,
     POOLS,
     REGISTRATION;

package body EVENTS is

     procedure ASSIGN_ELIMINATION
               (EVENT : in out EVENT_TYPE)
               is separate;
          --   If first time, get elimination type
          --   Get fencers in order of rank tree
          --   Build elimination list
          --   IF N_COMPETING = 8, order of ranks in list is
          --        1,8,5,4,3,6,7,2
          --   IF N_COMPETING = 16, order of ranks in list
          --        is
          --        1,16,9,8,5,12,13,4,
          --        3,14,11,6,7,1,10,15,2
          --   IF N_COMPETING = 32, order of ranks in list
          --        is
          --        1,32,17,16,9,24,25,8,
          --        5,28,21,12,13,20,29,4,
          --        3,30,19,14,11,22,27,6,
          --        7,26,23,10,15,18,31,2

     procedure ASSIGN_POOLS (EVENT : in out EVENT_TYPE)
               is separate;
          --   Get fencers in order of rank tree
          --   Build pool table

     procedure CLOSE_EVENT (EVENT : in out  EVENT_TYPE)
               is separate;

     procedure CORRECT_REGISTRATION
               (EVENT : in out EVENT_TYPE)
               is separate;
          --   Input fencer data
          --   modify fencer registration record

     procedure DELETE_FENCER
               (EVENT : in out EVENT_TYPE)
               is separate;
          --   Input fencer ID
          --   Delete from registration list

     procedure DISPLAY_ELIMINATION_ASSIGNMENTS
               (EVENT : in EVENT_TYPE)
               is separate;
```

```
procedure DISPLAY_POOL_ASSIGNMENTS
          (EVENT : in EVENT_TYPE)
          is separate;
     --   use pool tables
     --   get fencer data from registration

procedure GET_VALUES
          (EVENT  : in EVENT_TYPE;
           VALUES : out VALUE_TYPE)
          is separate;

procedure INPUT_ELIMINATION
          (EVENT : in out EVENT_TYPE)
          is separate;
     --   Input fencer results
     --   Get fencer registration record
     --   Modify fencer registration record
     --   Set rank indicator to FALSE
     --   This procedure needs expansion in order to
     --        express all requirements

procedure INPUT_POOL
          (EVENT : in out EVENT_TYPE)
          is separate;
     --   Input pool number and fencer data
     --   Get fencer registration record
     --   Modify fencer registration record
     --   Set rank indicator to FALSE
     --   This procedure needs expansion in order to
     --        express all requirements

procedure LOAD_EVENT (EVENT : in out  EVENT_TYPE)
          is separate;
     --   load registration table, pool table, and
     --        elimination  lists

procedure OPEN_EVENT (EVENT : in out  EVENT_TYPE)
          is separate;

procedure ORDER_REGISTRATION
          (EVENT : in out EVENT_TYPE)
          is separate;
     --   Input selection for name or rank order
     --   Get fencers in order of ID
     --   Build appropriate tree
     --   Name root points to name beginning with A
     --   Rank root points to rank 1
     --   If N_ROUNDS_POOLS = 0 and N_ROUNDS_ELIM = 0
     --        order by USFA class and point position
     --        rank 1 = class A fencer with point
     --              position 1
     --        highest rank number is class U
```

```
            --          for = class, lower rank number is lower
            --              ID
            --      else order by indicators
            --          rank 1 = highest V/B
            --          for = V/B fencers, use highest HS - HR
            --          if still = take lowest HR

        procedure REGISTER_FENCERS
                  (EVENT : in out EVENT_TYPE;
                   NEXT_ACTION : out REGISTRATION.ACTION_TYPE)
                  is separate;
            --    Input fencer data and NEXT_ACTION
            --    Validate club using club table
            --    Add fencer to registration list
            --    Do while NEXT_ACTION = "R "
            --    Save after reaching REGISTRATION.SAVE_POINT

        procedure REVIEW_POOLS
                  (EVENT : in EVENT_TYPE)
                  is separate;
            --    Display pool assignments
            --    Input commands to change display or
            --        move fencers
            --    Add fencers to pools or remove them

        procedure SAVE_EVENT (EVENT : in EVENT_TYPE)
                  is separate;

        procedure WITHDRAW_FENCER
                  (EVENT : in EVENT_TYPE)
                  is separate;
            --    Input fencer ID number
            --    Change fencer's STATUS to W

    end EVENTS;
```

```
--                    ****************************
--                    *                          *
--                    *       * FENCERS          *
--                    *                          *
--                    ****************************

with CONSTANTS;

package FENCERS is

     type FID_TYPE is range 0 .. CONSTANTS.MAX_FENCERS;
     type FNAME_TYPE is new STRING (1..20);
     type STRING4 is new STRING (1..4);
     type USFA_CLASS_TYPE is (A,B,C,D,E,U);
     type POINT_POSITION_TYPE is
          range 0 .. CONSTANTS.MAX_POINTS;
     type PROPORTION_TYPE is digits 4 range 0.0 .. 1.0;
     type POOL_NUMBER_TYPE is
          range 0 .. CONSTANTS.MAX_POOLS;
     type RANK_TYPE is range 0 .. CONSTANTS.MAX_FENCERS;
     type STATUS_TYPE is (C,E,W,K,Q,R);
          -- C = competing          K = rank, then eliminate
          -- E = eliminated         Q = rank, then withdraw
          -- W = withdrawn          R = include in repechage
round

     type FENCER_TYPE is
        record
          ID              :  FID_TYPE;
          NAME            :  FNAME_TYPE;
          UNATTACHED      :  BOOLEAN;
          MAJOR_CLUB      :  STRING4;
          CLUB_GROUP_1    :  STRING4;
               -- Affinity group for MAJOR_CLUB
          SECOND_CLUB     :  STRING4;
          CLUB_GROUP_2    :  STRING4;
               -- Affinity group for SECOND_CLUB
          USFA_CLASS      :  USFA_CLASS_TYPE;
          USFA_POSITION   :  POINT_POSITION_TYPE;
          VICTORIES       :  NATURAL;
          BOUTS           :  NATURAL;
          V_DIV_B         :  PROPORTION_TYPE;
          HITS_SCORED     :  NATURAL;
          HITS_RECEIVED   :  NATURAL;
          INDICATOR       :  INTEGER;
               -- INDICATOR = HITS_SCORED - HITS_RECEIVED
          POOL_NUMBER     :  POOL_NUMBER_TYPE;
          RANK            :  RANK_TYPE;
          STATUS          :  STATUS_TYPE;
        end record;

end FENCERS;
```

```
--                    ****************************
--                    *                          *
--                    *       * OPERATORS        *
--                    *                          *
--                    ****************************

package OPERATORS is

     type OPERATOR_ID_TYPE is new STRING (1 .. 4);
     type ACTION_TYPE is new STRING (1 .. 2);
     type PASSWORD_TYPE is new STRING (1 .. 8);
     type PERMISSION_TYPE is (R,U,S);
          --  R = display only
          --  U = can update tables
          --  S = can update security table

     procedure LOAD_SECURITY_TABLE;

     procedure OPERATOR_PASSWORD_CHANGE
               (OP_ID : in OPERATOR_ID_TYPE;
               NEW_PASSWORD : in PASSWORD_TYPE);

     procedure SECURITY_MENU
               (NEXT_ACTION : out ACTION_TYPE);

     procedure VALIDATE_OPERATOR
               (OP_ID : in OPERATOR_ID_TYPE;
               OP_PASSWORD : in PASSWORD_TYPE;
               OP_PERMISSION :  out PERMISSION_TYPE);
          --   If permission is blank, no permission
          --        because operator is not in table

end OPERATORS;

with CONSTANTS;
package body OPERATORS is

     type OPERATOR_NAME_TYPE is new STRING (1 .. 20);

     type OPERATOR_TYPE  is
       record
          OPERATOR_ID          :  OPERATOR_ID_TYPE;
          OPERATOR_NAME        :  OPERATOR_NAME_TYPE;
          PASSWORD             :  PASSWORD_TYPE;
          PERMISSION           :  PERMISSION_TYPE;
       end record;
```

```
          ------------------------------------------
          --                                      --
          --        OPERATORS GLOBAL DATA         --
          --                                      --
          ------------------------------------------

SECURITY_TABLE : array   (1 .. CONSTANTS.MAX_OPERATORS)
                         of OPERATOR_TYPE;
     --   in order by user ID

          ------------------------------------------

procedure ADD_OPERATORS is separate;
     --   Input operator data
     --   Add operator to security table

procedure DELETE_OPERATOR
          (OP_ID:  in OPERATOR_ID_TYPE)
          is separate;
     --   Input operator data
     --   Add operator to security table

procedure DISPLAY_SECURITY_TABLE is separate;

procedure LOAD_SECURITY_TABLE is separate;

procedure MODIFY_OPERATOR
          (OP_ID:  in OPERATOR_ID_TYPE)
          is separate;

procedure OPERATOR_INQUIRY
          (OP_ID : in OPERATOR_ID_TYPE;
          OPERATOR : out OPERATOR_TYPE)
          is separate;

procedure OPERATOR_PASSWORD_CHANGE
          (OP_ID : in OPERATOR_ID_TYPE;
          NEW_PASSWORD : in PASSWORD_TYPE)
          is separate;

procedure SAVE_SECURITY_TABLE is separate;

procedure SECURITY_MENU
          (NEXT_ACTION : out ACTION_TYPE)
          is separate;
```

```
     procedure VALIDATE_OPERATOR
               (OP_ID : in OPERATOR_ID_TYPE;
               OP_PASSWORD : in PASSWORD_TYPE;
               OP_PERMISSION :  out PERMISSION_TYPE)
               is separate;
          --   If permission is blank, no permission
          --        because operator is not in table

end OPERATORS;
```

```
--                    ****************************
--                    *                          *
--                    *         * POOLS          *
--                    *                          *
--                    ****************************

with CONSTANTS,
     FENCERS;

package POOLS is

     type POOL_SIZE_TYPE is
                    range 0 .. CONSTANTS.MAX_IN_POOL;

     function POOL_SIZE
               (POOL_NO : in FENCERS.POOL_NUMBER_TYPE)
               return POOL_SIZE_TYPE;

     procedure GET_FROM_POOL
               (POOL_NO   : in  FENCERS.POOL_NUMBER_TYPE;
               POSITION   : in  INTEGER;
               FENCER_ID  : out FENCERS.FID_TYPE);
          --   Return ID of pool member

     procedure INITIALIZE_TABLE;

     procedure INSERT_IN_POOL
               (POOL_NO   : in FENCERS.POOL_NUMBER_TYPE;
                FENCER_ID : in FENCERS.FID_TYPE);

     procedure LOAD_POOL_TABLE (EVENT_NUMBER : in NATURAL);

     procedure REMOVE_FROM_POOL
               (POOL_NO   : in FENCERS.POOL_NUMBER_TYPE;
                FENCER_ID : in FENCERS.FID_TYPE);

     procedure SAVE_POOL_TABLE (EVENT_NUMBER : in NATURAL);

end POOLS;

with CONSTANTS, FENCERS;
package body POOLS is

     type POOL_TYPE is array (1 .. CONSTANTS.MAX_IN_POOL)
                              of FENCERS.FID_TYPE;
```

```
   type POOL_REC_TYPE is
     record
        POOL           :  POOL_TYPE;
        POOL_SIZE      :  POOL_SIZE_TYPE;
        POOL_INDEX     :  POOL_SIZE_TYPE;
             -- next unfilled position
     end record;

             ------------------------------------------
             --                                      --
             --        POOLS GLOBAL DATA             --
             --                                      --
             ------------------------------------------

   POOL_TABLE     : array (1 .. CONSTANTS.MAX_POOLS)
                               of POOL_REC_TYPE;

             ------------------------------------------

   function  POOL_SIZE
             (POOL_NO : in FENCERS.POOL_NUMBER_TYPE)
             return POOL_SIZE_TYPE
             is separate;

   procedure GET_FROM_POOL
             (POOL_NO   : in  FENCERS.POOL_NUMBER_TYPE;
             POSITION   : in  INTEGER;
             FENCER_ID  : out FENCERS.FID_TYPE)
             is separate;
        --   Return ID of pool member

   procedure INITIALIZE_TABLE is separate;

   procedure INSERT_IN_POOL
             (POOL_NO   : in FENCERS.POOL_NUMBER_TYPE;
             FENCER_ID : in FENCERS.FID_TYPE)
             is separate;

   procedure LOAD_POOL_TABLE (EVENT_NUMBER : in NATURAL)
             is separate;

   procedure REMOVE_FROM_POOL
             (POOL_NO   : in FENCERS.POOL_NUMBER_TYPE;
             FENCER_ID : in FENCERS.FID_TYPE)
             is separate;

   procedure SAVE_POOL_TABLE (EVENT_NUMBER : in NATURAL)
             is separate;

end POOLS;
```

```
--                    ****************************
--                    *                          *
--                    *     * REGISTRATION       *
--                    *                          *
--                    ****************************

with CONSTANTS,
     FENCERS;

package REGISTRATION is

     type ACTION_TYPE is new STRING (1 .. 2);

     procedure ADD_FENCER
               (FENCER        : in FENCERS.FENCER_TYPE;
                NEXT_ACTION   : out ACTION_TYPE);

     procedure BUILD_NAME_INDEX
               (ROOT: out FENCERS.FID_TYPE);

     procedure BUILD_RANK_INDEX
               (BASIS          : in CHARACTER; -- C or I
                REPECHAGE      : in BOOLEAN;
                ROOT           : out FENCERS.FID_TYPE);
               --   BASIS = C: rank on USFA classification
               --   BASIS = I: rank on indicators

     procedure DELETE_FENCER (ID   : in FENCERS.FID_TYPE);

     procedure GET_FENCER
               (ID      :  in  FENCERS.FID_TYPE;
                FENCER  :  out FENCERS.FENCER_TYPE);

     procedure INITIALIZE_REG_TABLE;

     procedure LOAD_REG_TABLE
               (EVENT_NUMBER : in NATURAL);

     procedure MODIFY_FENCER
               (FENCER   : in FENCERS.FENCER_TYPE);

     procedure READ_BY_NAME
               (ROOT     : in out FENCERS.FID_TYPE;
               FENCER    : out FENCERS.FENCER_TYPE);

     procedure READ_BY_RANK
               (ROOT     : in out FENCERS.FID_TYPE;
               FENCER    : out FENCERS.FENCER_TYPE);
```

```
     procedure SAVE_REG_TABLE
               (EVENT_NUMBER : in NATURAL);

end REGISTRATION;

with CONSTANTS,
     FENCERS;
package body REGISTRATION is

     type REG_ITEM_TYPE is
       record
          REG_FENCER           :  FENCERS.FENCER_TYPE;
          RANK_LEFT_PTR        :  FENCERS.FID_TYPE;
               -- rank index
          RANK_CENTER_PTR      :  FENCERS.FID_TYPE;
          RANK_RIGHT_PTR       :  FENCERS.FID_TYPE;
          NAME_LEFT_PTR        :  FENCERS.FID_TYPE;
               -- name index
          NAME_RIGHT_PTR       :  FENCERS.FID_TYPE;
       end record;

               ------------------------------------------
               --                                      --
               --     REGISTRATION GLOBAL DATA         --
               --                                      --
               ------------------------------------------

     REG_TABLE :  array  (1 .. CONSTANTS.MAX_FENCERS)
                         of REG_ITEM_TYPE;

               ------------------------------------------

     procedure ADD_FENCER
               (FENCER         : in FENCERS.FENCER_TYPE;
                NEXT_ACTION    : out ACTION_TYPE)
               is separate;

     procedure BUILD_NAME_INDEX (ROOT: out FENCERS.FID_TYPE)
               is separate;

     procedure BUILD_RANK_INDEX
               (BASIS          : in CHARACTER; -- C or I
                REPECHAGE      : in BOOLEAN;
                ROOT           : out FENCERS.FID_TYPE)
               is separate;
               --   If ranking on indicators,
               --        rank status C, K, and Q
               --        reset K to E and Q to W
               --   If this is a repechage round,
               --        rank status R only
               --        reset R to C
```

```
procedure COMPARE_FENCERS
          (ID_1      :  in  FENCERS.FID_TYPE;
          ID_2       :  in  FENCERS.FID_TYPE;
          BASIS      :  in  CHARACTER;
          RESULT     :  out CHARACTER)
          is separate;
          --   RESULT = 'G'
          --        if fencer with ID_1 is greater
          --   RESULT = 'E' if fencers are equal
          --   RESULT = 'L'
          --        if fencer with ID_1 is less than
          --        fencer with ID_2

procedure DELETE_FENCER (ID   : in FENCERS.FID_TYPE)
          is separate;

procedure GET_FENCER
          (ID        :  in  FENCERS.FID_TYPE;
          FENCER     :  out FENCERS.FENCER_TYPE)
          is separate;

procedure INITIALIZE_REG_TABLE is separate;

procedure INSERT_ASCENDING_ORDER
          (ID  : in FENCERS.FID_TYPE)
          is separate;
          --   insert ID number in rank index tree
          --   rank index is built in ascending order
          --   fencers are compared on USFA
          --    classification

procedure INSERT_DESCENDING_ORDER
          (ID  : in FENCERS.FID_TYPE)
          is separate;
          --   insert ID number in rank index tree
          --   rank index is built in descending order
          --   fencers are compared on indicators

procedure LOAD_REG_TABLE
          (EVENT_NUMBER : in NATURAL)
          is separate;

procedure MODIFY_FENCER
          (FENCER   : in FENCERS.FENCER_TYPE)
          is separate;

procedure RANK_ON_USFA_CLASS is separate;

procedure RANK_ON_INDICATORS is separate;
```

```
      procedure READ_BY_NAME
                (ROOT      : in out  FENCERS.FID_TYPE;
                FENCER     : out FENCERS.FENCER_TYPE)
                is separate;
                --   return lowest name in alpha order (A)
                --   root = ID of next fencer for read

      procedure READ_BY_RANK
                (ROOT      : in out FENCERS.FID_TYPE;
                FENCER     : out FENCERS.FENCER_TYPE)
                is separate;
                --   return rank 1 fencer first
                --   root = ID of next fencer for read

      procedure SAVE_REG_TABLE
                (EVENT_NUMBER : in NATURAL)
                is separate;

      procedure  UPDATE_RANKS is separate;
                --   Access fencers in registration table in
                --        order of rank index
                --   Update rank FENCER.RANK in each record
                --        with rank number

end REGISTRATION;
```

```
--                    ****************************
--                    *                          *
--                    *         * REPORTS        *
--                    *                          *
--                    ****************************

with COMPETITION,
     FENCERS,
     REGISTRATION,
     TEXT_IO;

package REPORTS is

     procedure SPECIFY_REPORT;
          --   Input type of report
          --   produce report

end REPORTS;


with COMPETITION,
     FENCERS,
     REGISTRATION,
     TEXT_IO;

package body REPORTS is

     procedure REPORT_BY_ID is separate;

     procedure REPORT_BY_NAME is separate;

     procedure REPORT_BY_RANK is separate;

     procedure SPECIFY_REPORT is separate;

end REPORTS;
```

```
--                    ****************************
--                    *                          *
--                    *      * SCORE_SHEETS      *
--                    *                          *
--                    ****************************

with COMPETITION,
     FENCERS,
     POOLS,
     ELIM_LISTS,
     REGISTRATION,
     TEXT_IO;

package SCORE_SHEETS is

     procedure PRINT_ELIMINATION_FORMS;

     procedure PRINT_POOL_FORMS;
          --   For each pool, place fencers from the
          --        same club group together at the top of
          --        the list offencers in the pool

end SCORE_SHEETS;
```

GLOSSARY

abstraction: A simplified description of an entity that emphasizes some details while suppressing others. [1]

abstract data type: A high level abstraction not directly available in the implementation language.

algorithm: How a function may be done; a procedure consisting of a given number of specified steps or processes that when done in a specified order can be guaranteed to yield a desired result. [2]

analysis: The systematic process of reasoning about a problem and its constituent parts to understand what is needed or what must be done. [3]

component: A logically cohesive, loosely coupled module that denotes a single abstraction.

control flow: the sequencing of operations performed within a system.

data flow: the movement of data into, through, and out of a system.

denotational specification: Representing an expression with a function. The meaning of a program can be represented by a function - P: input + program process = output.

design: A transformation of requirements into a description of the structure of a software system; "An orderly decomposition of the total system function into subfunctions. The first level of decomposition usually defines the system architecture. Next, these subfunctions are further decomposed into smaller subfunctions, and the process continues until each primitive function is simple enough to easily translate into an algorithm. The transformation of each function into an algorithm completes the design." [4]

extensibility of language: Provides a mechanism to create higher level abstractions. [5]

external system perspective (system recognition perspective): the user's point of view, which partitions a system according to the utility of its parts, tending to lack understanding of construction details. [6]

executable specification: A specification having a formal semantics which can be interpreted to produce an operational semantics. A prototype simulates the formal semantics using the operational semantics. [7]

formal development methods: Methods based upon underlying mathematical theories which allow behavioral properties of systems to be unambiguously stated and deduced by formal reasoning. [8]

formalization of a data structure: The second level design after the initial definition of the structure.

function: What is done; the correspondence between sets of input data (domains) and sets of output data having specified values (ranges). [9]

information hiding: Suppression of unnecessary details.

information system: A computer system for maintaining and accessing a pool of information on some aspect of the real world. It has at its core a database of facts, interrogated by users through queries and manipulated by application programs. [10]

internal system perspective (system generation perspective): the designer's point of view, which partitions a system according to the function of its parts, tending to lose sight of the system's objectives. [11]

life cycle model: The set of procedures, rules, tools and techniques used to develop a system. [12]

localized: Physical grouping of entities that are logically related.

method: a disciplined process for producing software. [13]

methodology: Cooperating collections of methods or, in a more general sense, a philosophical approach to software development. [14] A set of rules that aid a designer in obtaining a solution to a problem. [15]

module: A self-contained software unit that may be developed independently. [16]

modularity: A node within a system can be completely and unambiguously understood independent of all other nodes. [17]

object: An entity that has state, is characterized by the operations that it absorbs and initiates, and is an instance of a class of objects. [18]

object oriented development: Software design and implementation in which the decomposition of a system is based on the concept of an object. [19]

primitive type: an elementary tool that describes the structure of data as part of an implementation language.

representation specification: Specifics of representing an entity within a computer, e.g. EBCDIC or ASCII, packed decimal or binary.

semantics of a language: Specifies the meaning of syntactically correct constructs of the language.

specification: A definition of a software system.

subsystem: A logical collection of cooperating program units that is subservient to a higher system structure.

syntax of a language: Specifies the combinations of symbols that are in the language.

system: A logical collection of cooperating subsystems that constitutes a coherent executable application.

TBD construct: A construct used to denote an incomplete element of the design that is still "To Be Determined."

type: A characterization of a set of values and operations applicable to the set. [20]

validation: Assuring that a program complies with the system requirements (Am I building the right product?). [21]

verification: Testing to assure that a program meets its specification (what is intended - Am I building the product right?). [22]

NOTES

SECTION 1

[1] I. Wasserman and Steven Gutz, "The Future of Programming," Communications of the ACM, 25, 3 (March 1982), p. 205.

[2] K.S. Mendes, quoted by Beichter, Herzog and Petsch, "SLAN-4 A Software Specification and Design Language," IEEE Transactions on Software Engineering, SE-10, 2 (March 1984), p. 155.

[3] M. Hamilton and S. Zeldin, "The Functional Life Cycle Model and Its Automation. USE.IT," The Journal of Systems and Software, 3, 1 (March 1983), p 26.

[4] In "Managing the Development of Large Software Systems: Concepts and Techniques," according to Bertrand Meyer, "On Formalism in Specifications," IEEE Software, 2, 1 (January 1985), p. 7.

[5] Robert A. Weissensee, "An Interim Ada Based Preliminary and Detailed Program Design Language, Part I," Journal of Pascal, Ada, & Modula-2, 4, 4 (July/August 1985), p. 6.

[6] Ibid., pp. 5 - 7.

[7] Hamilton and Zeldin, op. cit., pp. 27,30.

[8] Herbert Weber and Hartmut Ehrig, "Specification of Modular Systems," IEEE Transactions on Software Engineering, SE-12, 7 (July 1986).

[9] P. Hsia, A.T. Yaung, and S.H. Jiam, "Requirements Clustering for Incremental Construction of Software Systems," Proceedings COMPSAC '86.

[10] Robert Balzer, Thomas E. Cheatham and Cordell Green, "Software Technology in the 1990's: Using a New Paradigm," Computer, 16, 11 (November 1983), p. 40.

[11] Gruia-Catalin Roman, "A Taxonomy of Current Issues in Requirements Engineering," Computer, 18, 4 (April 1985), p. 20.

[12] Ibid.

[13] Vaclav Rajlich, "Paradigms for Design and Implementation in Ada," Communications of the ACM, 28, 7 (July 1985).

NOTES

SECTION 2

[1] Douglas T. Ross and Kenneth E. Schoman, "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, SE-3, 1 (January 1977), p. 8.

[2] Stephen Lipka, "Some Issues in Requirements Specifications," Proceedings COMPSAC '86, p. 57.

[3] Nakao, Haruna, Komoda, and Kaji, "A Structural Approach to System Requirements Analysis of Information Systems," Proceedings COMPSAC '80, p. 208.

[4] William Rzepka and Yutaka Ohno, "Requirements Engineering Environments: Software Tools for Modeling User Needs," Computer, 18, 4 (April 1985), pp. 9-10.

[5] Ibid., p. 10.

[6] Nakao, Haruna, Komoda, and Kaji, op. cit., pp. 207-213.

[7] Douglas T. Ross, "Applications and Extensions of SADT," Computer, 18, 4 (April 1985), pp. 25-34.

[8] Ross and Schoman, op. cit., p. 13.

[9] Meyer, op. cit., p. 6-26.

[10] Roman, op. cit., pp. 17-20.

[11] Ibid., p. 19.

[12] Wasserman and Gutz, op. cit., p. 204.

[13] Richard Morton and Karl Freburger, "Toward Methodology for Future Specifications," Proceedings COMPSAC 80, p. 204.

[14] Roman, op. cit. p. 15.

[15] Ross and Schoman, op. cit., p. 7.

[16] Barry Boehm, "Verifying and Validating Software Requirements and Design Specifications," IEEE Software, 1, 1 (January 1984), pp. 76-80.

[17] Roman, op. cit., pp. 16-17.

NOTES

[18] Daniel Teichroew and Ernest A. Hershey, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, SE-3, 1 (January 1977), p. 42.

[19] Morton and Freburger, op. cit., pp. 201-202.

[20] Rzepka and Ohno, op. cit., p. 10.

SECTION 3

[1] T.L.C. Chen and C.L. Steimle, "Two Design Approaches Using the Ada Language," Conference Proceedings, IEEE Southeastcon '87, p. 72.

[2] Stephen S. Yau and Jeffey J.-P. Tsai, "A Survey of Software Design Techniques, IEEE Transactions on Software Engineering, SE-12, 6 (June 1986).

[3] Grady Booch, "Object-Oriented Design," Ada Letters, I, 3 (March/April 1982), p. 66.

[4] Yau and Tsai, op. cit., p. 716.

[5] A. Goldberg, quoted by Stowe Boyd, "Object-Oriented Design and Pamela: A Comparison of Two Design Methods for Ada," Ada Letters, VII, 4 (July/Aug 1987), p. 74.

[6] Boyd, op. cit., p. 74.

[7] Grady Booch, Software Engineering with Ada, (Menlo Park: Benjamin/Cummings, 1987),p. 49, and "Object-oriented Design," p. 66. The steps are slightly different in the two sources. See also Software Components With Ada, (Menlo Park: Benjamin/Cummings, 1987), p. 17, in which Booch states that the steps evolved from an approach first proposed by Abbott.

[8] Booch, "Object-Oriented Design," p. 66.

[9] Chen and Steimle, op. cit., p. 72.

[10] Booch, "Object-Oriented Design," p. 68.

[11] Booch, Software Components With Ada, p. 21.

[12] Ed Seidewitz and Mike Stark, "Toward a General Object-Oriented Software Development Methodology," Ada Letters, VII, 3 (July/Aug 1987).

[13] Booch, Software Components With Ada, pp. 27 and 28.

[14] Ibid., p. 17.

[15] Ibid., p. 18.

[16] Booch, "Object-Oriented Design," p. 67.

[17] Booch, Software Components With Ada, p. 22.

[18] Booch, "Object-Oriented Design," p. 73.

[19] Ibid., p. 65.

[20] Booch, Software Components With Ada, p. 18.

[21] Ibid., p. 22.

[22] Ibid., p. 31.

[23] Boyd, op. cit.,pp. 76-77.

[24] Yau and Tsai, op. cit., p. 719.

[25] Boyd, op. cit., pp. 76.

[26] Chen and Steimle, op. cit. p. 72-76.

SECTION 4

[1] S.J. Goldsack, ed.,4 Ada for Specification: Possibilities and Limitations, Cambridge: Cambridge University Press, 1985, p. 13.

[2] Ibid., p. 11.

[3] Yau and Tsai, op. cit., p. 717.

[4] Booch, Software Engineering With Ada, pp. 29-31.

[5] Booch, "Object-Oriented Design," p. 65.

SECTION 5

[1] D.W. Waugh, "Ada as a Design Language," IBM/FSD Software Engineering Exchange, 3, 1 (October 1980), p. 8.

[2] The following list of features is from Goldsack, op. cit., pp. 19-20.

[3] Roman, op. cit., p. 20.

NOTES

[4] Jean E. Sammet, Douglas W. Waugh, Robert W. Reiter, "PDL/Ada -- A Design Language Based on Ada, "ACM81 Conference Proceedings," p. 21.

[5] Brian A. Nejmeh and H.E. Dunsmore, "A Survey of Program Design Languages (PDL's)," Proceedings COMPSAC '86, p. 452.

[6] Nejmeh and Dunsmore, op. cit., p. 448.

[7] Peter G. Anderson, A Design Language Based in Ada, Rochester: Rochester Institute of Technology, 1982, p. 3.

[8] IEEE Recommended Practice for Ada As a Program Design Language. New York: The Institute of Electrical and Electronics Engineers, Inc., 1987, p. 12.

[9] IEEE Recommended Practice for Ada As a Program Design Language, New York: The Institute of Electrical and Electronics Engineers, Inc., 1987.

[10] Rodney M. Bond, "Ada As A Program Description Language (PDL): A Project Management Perspective, Ada Letters, IV, 1 (July/Aug 1984), pp. 69,72-73.

SECTION 6

[1] Robert Balzer, Neil Goldman and David Wile, "Informality in Program Specifications," IEEE Transactions on Software Engineering, SE-4, 2 (March 1978), pp. 94-95.

[2] Nejmeh and Dunsmore, op. cit., p. 448.

[3] Boyd, op. cit., pp. 68.

[4] Nejmeh and Dunsmore, op. cit., p. 447.

[5] Booch, Software Components With Ada, p. 573.

[6] C. Beierle et al., "Integrated Program Development and Verification," Proceedings of the Symposium on Software Validation, 1983, p. 189.

[7] Wasserman and Gutz, op. cit., p. 204.

[8] John E. Gaffney, "Maximize Design Effort and Minimize Program Control Complexity - To Maximize Software Development Productivity," Proceedings COMPSAC 80, p. 226.

[9] IORL is discussed in section 7.7.

[10] M.I. Jackson, "Developing Ada Programs Using the Vienna Development Method (VDM)," Software Practice and Experience, 15, 3 (March 1985), p. 305. VDM is discussed n section 7.8.

[11] Balzer, Goldman and Wile, op. cit., p. 96.

[12] Stephen W. Smoliar and David Barstow, "Who Needs Programming Languages, and Why Do They Need Them?" ACM SIGPLAN Notices, 18, 6 (June 1983), pp. 149-150.

[13] Balzer, Goldman and Wile, op. cit., p. 97.

[14] Smoliar and Barstow, op. cit., pp. 150-156.

[15] Balzer, Goldman and Wile, op. cit., pp. 94-103.

[16] James R. Comer, "An Experimental Natural-Language Processor for Generating Data Type Specifications," ACM SIGPLAN Notices, 18, 12 (December 1983), pp. 25-33.

[17] Roman, op. cit., pp. 17-18.

[18] Alexander Borgida, "Features of Languages for the Development of Information Systems at the Conceptual Level," IEEE Software, 2, 1 (January 1985), p. 65.

[19] Roman, op. cit., p. 18.

SECTION 7

[1] Alan M. Davis, "Automating the Requirements Phase: Benefits to Later Phases of the Software Life-Cycle," Proceedings COMPSAC 80, p. 43.

[2] M. Hamilton and S. Zeldin, "The Functional Life Cycle Model and Its Automation. USE.IT.," The Journal of Systems and Software, 3, 1 (March 1983), p. 26.

[3] James Martin, System Design From Provably Correct Constructs, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1985, pp. 325-327.

[4] Hamilton and Zeldin, op.cit., p 40.

[5] Ibid., pp. 40-43.

[6] Stephen H. Caine and E. Kent Gordon, "PDL - A Tool for Software Design," Proceedings of the 1975 AFIPS National Computer Conference, 1975, p. 276.

[7] Nejmeh and Dunsmore, op. cit., p. 449.

NOTES

[8] PDL/81 Design Language Reference Guide, Pasadena, California: Caine, Farber & Gordon, Inc., p. 1.

[9] Nejmeh and Dunsmore, op. cit., p. 449.

[10] Beichter, Herzog and Petsch, op. cit., p. 156.

[11] Ibid., p. 161.

[12] Ibid., p. 187.

[13] Mack Alford, "SREM at the Age of Eight," Computer, 18 4 (April 1985), p. 39.

[14] Ibid., p. 39.

[15] M.W. Alford, "Software Engineering Requirements Engineering Methodology (SREM) at the Age of Four," Proccedings COMPSAC 80, p. 867.

[16] Alford, "SREM at the Age of Eight," p. 39.

[17] Alford, "Software Engineering Requirements Engineering Methodology (SREM) at the Age of Four," p. 866.

[18] Alford, "SREM at the Age of Eight," p. 37.

[19] Ibid., p. 39.

[20] Ibid., pp. 40-42.

[21] Ibid., p. 43-45.

[22] Paul A. Scheffer and Albert H. Stone,"A Case Study of SREM," Computer, 18 4 (April 1985), p. 50.

[23] Alford, "SREM at the Age of Eight," p. 40.

[24] Ibid., p. 40.

[25] Werner Trattnig and Helmut Kerner, "EDDA, A Very-High-Level Programming and Specification Language in the Style of SADT," Proceedings COMPSAC 80, p. 437.

[26] Ibid., pp. 436, 438.

[27] Ibid., p. 442.

[28] Teichroew and Hershey, op. cit., p. 46.

[29] Ibid., p. 44.

[30] Ibid., p. 47.

[31] Ibid., p. 45.

[32] Gene E. Sievert and Terrence A. Mizell, "Specification-Based Software Engineering With TAGS," Computer, 18, 4 (April 1985), p. 58.

[33] Ibid., p. 59.

[34] Ibid., p. 60.

[35] Ibid., p. 63.

[36] M.I. Jackson, "Developing Ada Programs Using the Vienna Development Method (VDM)," Software Practice and Experience, 15, 3 (March 1985), p. 317.

[37] Ibid., p. 313.

[38] Ibid., p. 317.

[39] M. Goedicke, "The Use of Formal Requirements Specifications in EDE in a Software Development Environment," Proceedings COMPSAC '86, p. 192.

[40] Ibid., p. 191.

[41] Ibid., p. 193.

[42] Ibid., p. 194.

[43] Ibid., p. 194.

[44] Marvin V. Zelkowitz and James Lyle, "Implementation of Program Specifications," Proceedings COMPSAC 80, p. 194.

[45] Ibid., p. 194.

[46] Ibid., p. 197.

[47] Davis, op. cit., p. 43.

[48] Ibid., p. 44.

[49] Ibid., p. 46.

[50] Ibid., p. 46.

[51] Ibid., p. 47.

[52] Louis E. Boydstun, Daniel Teichroew, Steven Spewak, Yuzo Yamamoto, Guy Starner, "Computer Aided Modeling of Information Systems," Proceedings COMPSAC 80, p. 39.

[53] Ibid., p. 39.

[54] Ibid., p. 39.

[55] O. Maurel and C. Bonnet, "Ada As a Program Design Language for a Telematic Services Project," Conference Proceedings, IEEE Computer Society 1984 Conference on Ada Applications and Environments, pp. 89-90.

SECTION 8

[1] Richard Platek, "The Use of Ada As An Implementation Language in Formally Specified Systems," Proceedings of the 1984 Symposium on Security and Privacy, 1984, p. 107.

[2] Booch, Software Engineering With Ada, p. 415.

[3] Sammet, Waugh, and Reiter, op. cit., pp. 21-22.

[4] Ibid., p. 22.

[5] Booch, Software Engineering With Ada, pp. 419-420.

[6] George Corliss, "Using Ada as a Design Language," Proceedings of the 1983 ACM Annual Conference, 1983, p. 70.

[7] Booch, Software Engineering With Ada, p. 29.

[8] Ibid., p. 50.

[9] Hal Hart, "Ada For Design: An Approach for Transitioning Industry Software Developers," Ada Letters, II, 1 (July/August 1982), p. 54.

[10] Corliss, op. cit., p. 69.

[11] Maurel and Bonnet, op. cit., p. 91.

[12] Corliss, op. cit., p. 69.

[13] Peter G. Anderson, A Design Language Based on Ada. Rochester: Rochester Institute of Technology, 1982, p. 5.

[14] Maurel and Bonnet, op. cit., p. 91.

NOTES

[15] Ibid., p. 90.

[16] Booch, Software Engineering With Ada, p. 399.

[17] J.P. Alstad,"Problems With Ada as a Program Design Language: A Position Paper," Ada Letters, II, 6 (May/June 1983), p. 52.

[18] Platek, op. cit., p. 107.

[19] Alstad, op. cit., p. 51.

[20] Goldsack, op. cit., pp. 34 and 44.

[21] Sammet, Waugh, and Reiter, op. cit., p. 21.

[22] Larry Yelowitz,"Practical Experience with an Ada-Based Formal Specification Language on a Large Project," Proceedings of the 1984 Symposium on Security and Privacy, 1984, p.112.

[23] David C. Luckham, "On the Design of Anna, A Specification Language for Ada," Proceedings of the Symposium on Software Validation, 1983, p. 209.

[24] Ibid., p. 209.

[25] Ibid., p. 212.

[26] Ibid., pp. 215-216.

[27] Ibid., p. 216.

[28] Ibid., p. 224.

[29] David C. Luckham and Friedrich von Henke, "An Overview of Anna, a Specification Language for Ada," IEEE Software, 2, 2 (March 1985), p. 17.

[30] Luckham, op. cit., p. 222. For a discussion of Clear, another language based on category theory,See Steven D. Litvintchouk and Allen S. Matsumoto, "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification," IEEE Transaction on Software Engineering, SE-10, 5 (September 1984).

[31] Luckham and von Henke, op. cit., p. 19.

[32] Luckham, op. cit., pp. 214-215.

[33] Luckham and von Henke, op. cit., p. 10.

[34] Ibid., p. 21.

[35] Luckham, op. cit., p. 209.

[36] Goldsack, op. cit., p. 143.

[37] Ibid., p. 246.

[38] Ibid., pp. 216-217.

[39] Ibid., p. 217.

[40] Ibid., p. 217.

[41] Ibid., p. 219.

[42] Ibid., p. 220.

[43] Ibid., p. 228.

[44] Nejmeh and Dunsmore, op. cit., p. 457.

[45] Yelowitz, op. cit., p. 112.

[46] Nejmeh and Dunsmore, op. cit., p. 451.

[47] Ibid., p. 451.

[48] Ibid., p. 452.

[49] Yelowitz, op. cit., p. 112.

[50] Nejmeh and Dunsmore, op. cit., p. 451.

[51] Ibid., p. 451.

[52] Nejmeh and Dunsmore, op. cit., p. 450.

[53] Hart, op. cit., p. 55.

[54] Ibid., p. 55.

[55] Nejmeh and Dunsmore, op. cit., p. 450.

[56] Hart, op. cit., p. 56.

[57] Sammet, Waugh, and Reiter, op. cit., p. 22.

[58] Ibid., p. 19.

[59] Waugh, op. cit., p. 9.

[60] Ibid., p. 8.

[61] Ibid., p. 9.

[62] Sammet, Waugh, and Reiter, op. cit., pp. 23-25.

[63] Ibid., p. 25.

[64] Nejmeh and Dunsmore, op. cit., p. 450.

[65] Yelowitz, op. cit., p. 112.

[66] Yelowitz, op. cit., p. 112.

[67] Nancy Linden Yavne, "A Simple Approach To A Relaxed Syntax For An Ada PDL, Ada Letters, V, 1 (July/August 1985), p. 71.

[68] Ibid., p. 75.

[69] Saro B. Ghazarian, "The Allure of Ada for Programmers, Mini-Micro Systems, XX, 10 (October 1987), p. 137.

[70] Ibid., p. 137.

[71] Ibid., p. 135.

[72] Ibid., p. 137.

[73] IEEE Recommended Practice for Ada As a Program Design Language, p. 11.

[74] R.J.A. Buhr, System Design With Ada, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984, pp. 38-43.

[75] Booch, Software Engineering With Ada, p. 467.

[76] Ibid., p. 468.

[77] Booch, Software Components With Ada, p. 579.

[78] Booch, Software Engineering With Ada, p. 465.

[79] Ibid., p. 467.

[80] Maurel and Bonnet, op. cit., p. 90.

[81] Ibid., p. 90.

[82] Booch, Software Engineering With Ada, p. 417.

[83] Ibid., p. 418.

[84] Anderson, op. cit., p. 8.

[85] Ibid., p. 9.

[86] Bryce M. Bardin, "A 'To Be Determined' Package for Ada Development," Ada Letters, V, 3 (Nov/Dec 1985), p. 47.

[87] Ibid., p. 50.

[88] Ibid., p. 51.

[89] Robert A. Weissensee, "An Interim Ada Based Preliminary and Detailed Program Design Language, Part III," Journal of Pascal, Ada, & Modula-2, 5, 1 (January/February 1986), p. 6.

[90] Robert A. Weissensee, "An Interim Ada Based Preliminary and Detailed Program Design Language, Part II," Journal of Pascal, Ada, & Modula-2, 4, 5 (September/October 1985), p. 9.

[91] Eran Gabber, "The Middle Way Approach for Ada Based PDL Syntax," Ada Letters, II, 4 (Jan/Feb 1983), p. 65.

[92] Ibid., p. 65.

[93] Ibid., p. 66.

[94] I.C. Pyle, "A Package for Specifying Ada Programs," Ada Letters, III, 5 (March/April 1984), pp. 65-66.

[95] Ibid., p. 68.

[96] Ibid., p. 68.

SECTION 9

[1] Grady Booch, Software Engineering With Ada.

[2] Ibid., p. 48.

[3] Booch, "Object-Oriented Design," p. 73.

[4] Booch, Software Engineering With Ada, p. 146.

[5] Process Albums Data Base example. Ibid., chapters 9 and 12.

[6] Ibid., p. 56.

[7] Ibid., p. 59.

[8] R.J.A. Buhr, op. cit., chapter 3.

[9] Booch, Software Engineering With Ada, p. 48. Also "Object Oriented Design,"p. 66.

[10] Booch, Software Engineering With Ada. Booch did not imply that these implementations were part of the design. They illustrate the result of the design.

SECTION 11

[1] Nejmeh and Dunsmore, op. cit., p. 450.

[2] Denis P. Geller, "B-ware -- Contradictions in a Software Development Plan, ACM SIGSOFT Software Engineering Notes, 10, 1 (Jan 19895), pp. 49-50.

[3] Lawrence M. Lindley, "Ada Programming Design Language Survey," Ada Letters, II, 3 (Nov/Dec 1982), p. 33.

[4] Nejmeh and Dunsmore, op. cit., p. 450.

[5] IEEE Recommended Practice for Ada As a Program Design Language, p. 11.

[6] Booch, Software Components With Ada, p. 11.

[7] Litvintchouk and Matsumoto, op. cit., p. 544.

[8] Ibid., p. 548.

[9] Ibid., p. 550.

[10] Ibid., p. 545.

[11] Ibid., pp. 545-546.

[12] Wasserman and Gutz, op. cit., p. 203.

[13] Ibid., p. 203.

[14] Waugh, op. cit., p. 8.

[15] Ghazarian, op. cit., p. 137.

NOTES

GLOSSARY

[1] Booch, Software Components With Ada, p. 16.

[2] Ned Chapin, "Semi-Code in Design and Maintenance," Computers and People, 27, 6 (June 1978), p. 17.

[3] Rzepka and Ohno, op. cit., p. 9.

[4] Davis, op. cit., p. 42.

[5] Booch, "Object-Oriented Design," p. 67.

[6] Hsia, Yaung, and Jiam, op. cit., p. 204.

[7] Goedicke, op. cit., p. 194.

[8] Jackson, op. cit., pp 305-306.

[9] Chapin, op. cit., p. 17.

[10] Borgida, op. cit., p. 63.

[11] Hsia, Yaung, and Jiam, op. cit., p. 204.

[12] Hamilton and Zeldin, op.cit., p 25.

[13] Booch, Software Engineering With Ada, p. 36.

[14] Ibid., p. 36.

[15] Hamilton and Zeldin, op.cit., p 32.

[16] Weber and Ehrig, op. cit., p. 785.

[17] C.R. Everhart, "A Unified Approach to Software (System) Engineering," Proceedings COMPSAC 80, p 51.

[18] Booch, Software Engineering With Ada, p. 51.

[19] Booch, Software Components With Ada, p. 12.

[20] Booch, "Object-Oriented Design," p. 69.

[21] Boehm, op. cit., p. 75.

[22] Ibid., p. 75.

BIBLIOGRAPHY

Alford, M.W. "Software Requirements Engineering Methodology (SREM) at the Age of Four," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 866-874.

Alford, Mack. "SREM at the Age of Eight," Computer, 18, 4 (April 1985): 36-46.

Alstad, J.P. "Problems With Ada as a Program Design Language: A Position Paper", Ada Letters, II, 6 (May/June 1983): 51-52.

Anderson, Peter G. A Design Language Based on Ada. Rochester: Rochester Institute of Technology, 1982.

Balzer, Robert; Cheatham, Thomas E.; and Green, Cordell. "Software Technology in the 1990's: Using a New Paradigm," Computer, 16, 11 (November 1983): 39-45.

Balzer, Robert; Goldman, Neil; and Wile, David. "Informality in Program Specifications," IEEE Transactions on Software Engineering, SE-4, 2 (March 1978): 94-103.

Bardin, Bryce M. "A 'To Be Determined' Package for Ada Development," Ada Letters, V, 3 (Nov/Dec 1985): 45-56.

Beichter, Friedrich; Herzog, Otthein; and Petsch, Heiko. "SLAN-4 - A Software Specification and Design Language," IEEE Transactions on Software Engineering, SE-10, 2 (March 1984): 155-206.

Beierle, C.; Gerlach, M.: Gobel, R.; Olthoff, W.; Raulefs, P.; and Voss, A. "Integrated Program Development and Verification," Proceedings of the Symposium on Software Validation, 1983, 189-205.

Boehm, Barry W. "Verifying and Validating Software Requirements and Design Specifications," IEEE Software, 1, 1 (January 1984): 75-88.

Bond, Rodney M. "Ada As A Program Description Language (PDL): A Project Management Perspective," Ada Letters, IV, 1 (July/Aug 1984): 67-73.

Booch, Grady. "Describing Software Design in Ada," ACM SIGPLAN Notices 16, 9 (September 1981): 42-47.

Booch, Grady. "Object-oriented Design," Ada Letters, I, 3 (March/April 1982): 64-76.

Booch, Grady. Software Components With Ada. Menlo Park, California: Benjamin/Cummings Publishing Company, Inc., 1987.

Booch, Grady. Software Engineering With Ada. Menlo Park, California: Benjamin/Cummings Publishing Company, Inc., 1987.

Borgida, Alexander. "Features Of Languages for the Development of Information Systems at the Conceptual Level," IEEE Software, 2, 1 (January 1985): 63-72.

Boyd, Stowe. "Object-Oriented Design and PAMELA: A Comparison of Two Design Methods for Ada," Ada Letters, VII, 4 (July/Aug 1987): 68-78.

Boydstun, Louis E.; Teichroew, Daniel; Spewak, Steven; Yamamoto, Yuzo; and Starner, Guy. "Computer Aided Modeling of Information Systems," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 37-41.

Buhr, R.J.A. System Design With Ada. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.

Caine, Stephen H., and Gordon, E. Kent. "PDL - A Tool for Software Design," Proceedings of the 1975 AFIPS National Computer Conference, 1975, 271-276.

Chapin, Ned. "Semi-Code in Design and Maintenance," Computers and People, 27, 6 (June 1978): 17-27.

Chen, T.L.C., and Steimle, C.L. "Two Design Approaches Using the Ada Language," Conference Proceedings. IEEE Southeastcon '87. 1987 vol. 1: 72-76.

Comer, James R. "An Experimental Natural-Language Processor for Generating Data Type Specifications," ACM SIGPLAN Notices 18, 12 (December 1983): 25-33.

Corliss, George. "Using Ada as a Design Language - Classroom Experience," Proceedings of the 1983 ACM Annual Conference, 1983, 66-71.

Davis, Alan. "Automating the Requirements Phase: Benefits to Later Phases of the Software Life-Cycle," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference,
1980, 42-48.

Everhart, C.R. "A Unified Approach to Software (System) Engineering," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 49-55.

Gabber, Eran. "The Middle Way Approach for Ada Based PDL Syntax," Ada Letters, II, 4 (Jan/Feb 1983): 64-67.

Gaffney, John E. "Maximize Design Effort and Minimize Program Control Complexity - To Maximize Software Development Productivity," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 225-228.

Geller, Denis P. "B-ware -- Contradictions in a Software Development Plan," ACM SIGSOFT Software Engineering Notes, 10, 1 (Jan 1985): 48-51.

Ghazarian, Saro B. "The Allure of Ada for Programmers," Mini-Micro Systems, XX, 10 (October 1987): 135-137.

Goedicke, M. "The Use of Formal Requirements Specifications in EDE in a Software Development Environment," Proceedings COMPSAC '86, The IEEE Computer Society's Tenth Annual International Computer Software and Applications Conference, 1986, 190-196.

Goldsack, S.J., ed. Ada for Specification: Possibilities and Limitations. Cambridge: Cambridge University Press, 1985.

Hamilton, M. and Zeldin, S. "The Functional Life Cycle Model and Its Automation. USE.IT," The Journal of Systems and Software, 3, 1 (March 1983): 25-62.

Hart, Hal. "Ada for Design: An Approach for Transitioning Industry Software Developers," Ada Letters, II, 1 (July/August 1982): 50-57.

Hsia, P.; Yaung, A.T.; and Jiam, S.H. "Requirements Clustering for Incremental Construction of Software Systems," Proceedings COMPSAC '86, The IEEE Computer Society's Tenth Annual International Computer Software and Applications Conference, 1986, 204-211.

IEEE Recommended Practice for Ada As a Program Design Language. New York: The Institute of Electrical and Electronics Engineers, Inc., 1987 (IEEE Std. 990-1987).

IEEE Standard Glossary of Software Engineering Terminology. New York: The Institute of Electrical and Electronics Engineers, Inc., 1983 (IEEE Std. 729-1983).

Jackson, M.I. "Developing Ada Programs Using the Vienna Development Method (VDM)," Software Practice and Experience, 15, 3 (March 1985): 305-318.

Johnson, Philip I. The Ada Primer. New York: McGraw-Hill Book Company, 1985.

Kerner, Judy. "Ada DL Developers Matrix," Ada Letters, VII, 3 (May/June 1987): 107-116.

Lindley, Lawrence M. "Ada Program Design Language Survey," Ada Letters, II, 3 (Nov/Dec 1982): 32-33.

Lipka, Stephen. "Some Issues in Requirements Specifications," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 56-58.

Litvintchouk, Steven D. and Matsumoto, Allen S. "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification," IEEE Transactions on Software Engineering, SE-10, 5 (September 1984): 544-551.

Luckham, David C., and von Henke, Friedrich W. "An Overview of Anna, a Specification Language for Ada," IEEE Software, 2, 2 (March 1985): 9-22.

Luckham, David C. "On the Design of Anna, A Specification Language for Ada," Proceedings of the Symposium on Software Validation, 1983, 207-227.

Martin, James. System Design From Provably Correct Constructs, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1985.

Maurel, O. and Bonnet, C. "Ada As a Program Design Language for a Telematic Services Project," Conference Proceedings. IEEE Computer Society 1984 Conference on Ada Applications and Environments, 89-94.

Meyer, Bertrand. "On Formalism in Specifications," IEEE Software, 2, 1 (January 1985): 6-26.

Mitchell, John R. "Observations on the Use of Seven Structured Programming Techniques," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 229-235.

Morton, Richard and Freburger, Karl. "Toward Methodology for Future Specifications," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 201-206.

Nakao, Kazuo; Haruna, Koichi; Komoda, Norohisa; and Kaji, Hiroyuki. "A Structural Approach to System Requirements Analysis of Information Systems," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 207-213.

Nejmeh,Brian A., and Dunsmore, H.E. "A Survey of Program Design Languages (PDL's)," Proceedings COMPSAC '86, The IEEE Computer Society's Tenth Annual International Computer Software and Applications Conference, 1986, 447-455.

PDL/81 Design Language Reference Guide. Pasadena, California: Caine, Farber & Gordon, Inc., 1982.

Platek, Richard. "The Use of Ada As An Implementation Language in Formally Specified Systems," Proceedings of the 1984 Symposium on Security and Privacy, 1984, 107-110.

Pyle, I.C. "A Package for Specifying Ada Programs," Ada Letters, III, 5 (March/April 1984): 63-68.

Rajlich, Vaclav. "Paradigms for Design and Implementation in Ada," Communications of the ACM, 28, 7 (July 1985): 718-727.

Roman, Gruia-Catalin. "A Taxonomy of Current Issues in Requirements Engineering," Computer, 18, 4 (April 1985): 14-22.

Ross, Douglas T. and Schoman, Kenneth E. "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, SE-3, 1 (January 1977): 6-15.

Ross, Douglas T. "Applications and Extemsions of SADT," Computer, 18, 4 (April 1985): 25-34.

Rzepka, William and Ohno, Yutaka. "Requirements Engineering Environments: Software Tools for Modeling User Needs," Computer, 18, 4 (April 1985): 9-12.

Sammet, Jean E.; Waugh, D.W., and Reiter, R.W. "PDL/Ada -- A Design Language Based on Ada," ACM81 Conference Proceedings, 1981, 217-229.

Seidewitz, Ed and Stark, Mike. "Toward a General Object-Oriented Software Development Methodology," Ada Letters, VII, 3 (July/Aug 1987): 54-67.

Sievert, Gene E. and Mizell, Terrence A. "Specification-Based Software Engineering With TAGS," Computer, 18, 4 (April 1985): 56-65.

Scheffer, Paul A. and Stone, Albert H. "A Case Study of SREM," Computer, 18, 4 (April 1985): 47-54.

Smoliar, Stephen W. and Barstow, David. "Who Needs Programming Languages, and Why Do They Need Them? or No Matter How High the Level, It's Still Programming," ACM SIGPLAN Notices 18, 6 (June 1983): 149-157.

Teichroew, Daniel and Hershey, Ernest A. "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, SE-3, 1 (January 1977): 41-48.

Trattnig, Werner and Kerner, Helmut. "EDDA, A Very-High-Level Programming and Specification Language in the Style of SADT," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 436-443.

USE.IT System Reference Manual. Cambridge, Massachusetts: Higher Order Software, Inc., 1984.

Wasserman, I. and Gutz, Steven. "The Future of Programming," Communications of the ACM, 25, 3 (March 1982): 196-206.

Waugh, D.W. "Ada As A Design Language," IBM/FSD Software Engineering Exchange, Special Ada Version, 3,1 (October 1980): 8-12.

Waugh, Douglas; Reiter, Robert; and Sammet, Jean. PDL/ADA Syntax. Bethesda, Maryland: IBM Federal Systems Division, 1981.

Weber, Herbert and Ehrig, Hartmut. "Specification of Modular Systems," IEEE Transactions on Software Engineering, SE-12, 7 (July 1986): 784-798.

Weissensee, Robert A. "An Interim Ada Based Preliminary and Detailed Program Design Language, Part I," Journal of Pascal, Ada, & Modula-2, 4, 4 (July/August 1985): 1-8.

Weissensee, Robert A. "An Interim Ada Based Preliminary and Detailed Program Design Language, Part II," Journal of Pascal, Ada, & Modula-2, 4, 5 (September/October 1985): 5-12.

Weissensee, Robert A. "An Interim Ada Based Preliminary and Detailed Program Design Language, Part III," Journal of Pascal, Ada, & Modula-2, 5, 1 (January/February 1986): 3-13.

Yau, Stephen S.; Nicholl, Robin A.; and Tsai, Jeffery J.-P. "An Evolution Model for Software Maintenance," Proceedings COMPSAC '86, The IEEE Computer Society's Tenth Annual International Computer Software and Applications Conference, 1986, 440-446.

Yau, Stephen S. and Tsai, Jeffery J.-P. "A Survey of Software Design Techniques," IEEE Transactions on Software Engineering, SE-12, 6 (June 1986): 713-721.

Yavne, Nancy Linden. "A Simple Approach To A Relaxed Syntax For An Ada PDL," Ada Letters, V, 1 (July/August 1985): 71-78.

Yelowitz, Larry. "Practical Experience with an Ada-Based Formal Specification Language on a Large Project," Proceedings of the 1984 Symposium on Security and Privacy, 1984, 111-112.

Zelkowitz, Marvin V. and Lyle, James. "Implementation of Program Specifications," Proceedings COMPSAC 80. The IEEE Computer Society's Fourth International Computer Software and Applications Conference, 1980, 194-200.