(1)

A DECISION-BASED METHODOLOGY FOR OBJECT ORIENTED-DESIGN

THESIS

Patrick Denis Barnes
Captain, USAF

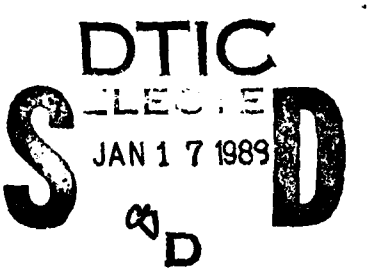
AFIT/GCS/ENG/88D-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

AFIT/GCS/ENG/88D-1

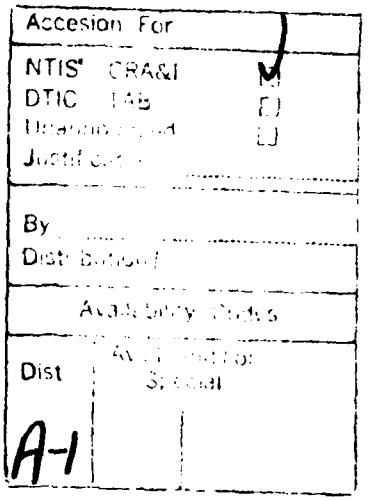


A DECISION-BASED METHODOLOGY FOR OBJECT ORIENTED-DESIGN

THESIS

Patrick Denis Barnes
Captain, USAF

AFIT/GCS/ENG/88D-1



AFIT/GCS/ENG/88D-1

A DECISION-BASED METHODOLOGY FOR OBJECT ORIENTED-DESIGN

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Science)

Patrick Denis Barnes, A.A.S, B.S.
Captain, USAF

December 16, 1988

Who am I, O Sovereign Lord, and what is my family that you have brought me this far?

King David, 2 Samuel 7:18

Just as David attributed his achievements and position to his Sovereign Lord, I want to give recognition first and foremost to my God who has brought me so far, and has made me who and what I am. What I've achieved through this effort is due primarily to the grace and mercy of a God who loved me enough to rescue me from my own selfish pursuits, and gave me the heart and desire to become all he wants me to be through the person of his Son Jesus Christ.

A special note of gratitude belongs to my wife Brenda. Her compassion, perseverance, sacrifice, and constant encouragement made possible the long nights at the computer and the many extra hours I was able to devote to this research. I also thank my four children: Jessica, Jeremy, James, and Jeanette for reminding me of the importance of making family a top priority in spite of a demanding work load.

I would also like to thank Mike T., Bill, Nick, Mark, Mike F., and Norm who, while working on similar efforts, were willing to take the time to meet together weekly for mutual encouragement and prayer. Their friendship, uncompromising standards, and lifestyle were personally challenging and added a special time of refreshing to an otherwise mentally and emotionally taxing environment.

Finally, I thank my advisor, Dr. Thomas C. Hartrum, and my committee members: Captain David Umphress and Captain Bruce George for allowing me the freedom to follow where the research seemed to lead rather than constraining my somewhat limited creativity. Their standards for integrity, quality, and perseverance pushed me to exceed my preconceived limitations and draw on previously untapped resources to finish this work.

Table of Contents

List of Figures

List of Tables

Abstract

The task of object-oriented development raises a new set of design problems. Specifically: how to scope a problem based on objects rather than functions; how to select the best objects; how to encapsulate data structures with the *right* set of operations; and when to stop decomposing a system into objects and begin describing the algorithms that implement those objects' behaviors. The difficulty of making these decisions is increased when the requirements documentation was not developed with an object-oriented paradigm in mind.

Within the past few years, several software development environments have been proposed or developed implementing an object-oriented design methodology. Many, however, are concerned only with "programming in the small" activities, or with providing capabilities for capturing, representing, and storing design decisions once they are made, rather than with helping the designer make sound design decisions. Recognizing the importance of supporting design decision making, the result of this study was formulation of a methodology for object-oriented design using the concepts of decision support systems.

This thesis describes an object-oriented design methodology based on the four problems or *decisions* stated above. These decisions are summarized as *analysis*, *identification*, *encapsulation*, and *decomposition*. An object model structure is also defined to provide a foundation for organizing design information. The object model is described by a set of database relations, and includes a three view graphic representation providing block, detail and control flow graphs.

A prototype design tool was implemented to evaluate the methodology. Software for the tool was developed using a PC based implementation of the Smalltalk Object-Oriented Programming Language. Maximum use was made of decision support system techniques such as concept-mapping, storyboarding, the hook book,

and adaptive design. As a decision support system, the tool provides the software developer with key requirements specification and software engineering qualitative information to aid in the judgement and design decision making process.

A DECISION-BASED METHODOLOGY FOR OBJECT ORIENTED-DESIGN

I. Introduction

Escalating software development and maintenance costs as well as demand for software solutions to increasingly complex problems have mandated new techniques for engineering reliable, maintainable computer software. One approach to improving software quality is the use of the *object-oriented* paradigm for design and programming. This thesis is concerned with the *design* problem.

1.1 Background

1.1.1 The Object-Oriented Paradigm. The term "object-oriented" probably became best known through the simulation and prototyping languages SIMULA and Smalltalk developed in the 1970s [37]. The object- oriented paradigm has been applied widely and can be seen in the techniques of a variety of current software application areas. Examples are the database entity-relationship-attribute model, the frame-based approach taken in artificial intelligence, and simulation methods employing a similar entity-attribute-activity model.

Simply put, to say a method is object-oriented is to say that its representation of the problem space consists primarily of objects and their related attributes and operations [10]. This approach may be contrasted with procedural, data-flow, or data-structure paradigms.

One of the major applications of the paradigm based on the Smalltalk research is object-oriented programming (OOP). OOP research has involved the development

of a new generation of languages based on *objects*, which are organized into *classes*, and *inherit* attributes and operations called *methods* from other objects [52]. Rather than relying exclusively on the logic constructs of sequence, selection, and iteration, the main control structure is the *message* which instructs an object to perform some method on its private data. While such languages have not yet entered the mainstream, the principles discovered and techniques employed in OOP may be applied to *design* of systems to be constructed using more general purpose languages.

1.1.2 Object-Oriented Design (OOD). The goal of design is to produce a model or representation of a system at a level of detail such that it can be built [37]. Thus a design of a software system is a model or representation in terms that will provide sufficient guidance and understanding to the programmer who will write the program. In addition to the end product, design can also be thought of as either the process or activity of designing. A methodology for design must therefore describe both the product and the activity; in our case, an object-oriented representation and the steps required to develop that representation.

Pressman [37] goes on to further describe the design process as combining

> ... intuition and judgement based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation.

Pressman's definition indicates that in addition to defining a model and some design steps, an environment is needed to support judgment and choice, embody design principles and/or heuristics, guide an iterative development process, and enable qualitative evaluation of the finished product.

Several methodologies have been proposed for an object-oriented approach to design. Examples are [19], [14], [11], [12], [42], and [2].

When an object-oriented approach is applied to the design of computer software, a clean, component level abstraction of the design solution is developed in

terms of problem space entities rather than data processing constructs. Actual data structures and algorithms are hidden within the objects and only their outward behavior and interrelationships are shown at the architectural level. The desirable software engineering principles of abstraction, information hiding, and modularity, are therefore directly embodied within or enhanced by the object-oriented paradigm [37].

OOD's ability to describe concurrent, complex, and abstract systems has generated considerable interest in the software engineering community. Of particular interest to the Department of Defense is the use of object-oriented design for improving the development, reliability, and maintainability of programs written in the object-based general purpose programming language—Ada [19].

1.2 Statement of the Problem

As a partial life-cycle methodology applying primarily to preliminary design [1] [37], OOD's lack of compatibility with established requirements and analysis techniques, such as SADT [40] and Structured Analysis [15], have left it virtually unused in an otherwise rapidly expanding Ada environment for the DOD. To get OOD out of the classroom and into the hands of developers, techniques must be developed for transitioning to OOD from these accepted analysis methods.

While several automated tools have been or are currently being developed for OOD [14] [22] [2], they are machine and language dependent, and unable to integrate efficiently into the users' existing environment. Conversely, while many programming support environments are extensible, they provide mostly implementation-oriented aids such as code generators and context-sensitive editors [32], rather than aids for the choices and decisions that must be made during design. Both extensibility and design decision support are needed.

[1] Virtually all OOD methodologies use conventional techniques for detail design such as psuedocode or flow diagrams.

Finally, since OOD methodologies have yet to be proven over a significant number or spectrum of software development efforts, it is impossible to say what the *right* or *best* OOD methodology is. A significant investment has been made by both the DOD and industry in building expensive software development environments. Many such products, based on unproven methodologies, are often little used or entirely abandoned. An environment with the capability to adapt to variations in the methodology is needed. An adaptive approach would take a *first cut* methodology, quickly prototype it, then modify it over a period of time until a proven technique is developed.

1.3 Scope

The problems of *transition*, *integration*, and *adaptation* discussed in the previous section are the focus of this effort. A methodology is presented for transitioning to an object-oriented design from a formal requirements specification. The methodology is implemented in a prototype environment which emphasizes software design phase decisions. The environment demonstrates the benefits of on-line access to the requirements analysis database information. Finally, the user interface is user adaptive—even to the extent of altering the OOD methodology.

This thesis is not a defense of object-oriented design, nor a proof of concept for the object-oriented approach in general. Rather, we assume the validity of the object-oriented and software engineering principles involved, and define and prototype a generic, language independent OOD methodology and environment, suitable for adaptation and research.

1.4 Research Approach

1.4.1 Model Definition. The first step to developing a transition methodology was to understand and define the requirements and design models to be used. A thorough review of the literature was performed to delineate generic design and

requirements specification models. A definition and representation of the object model which best applies to design was selected. The resulting model descriptions and definitions are presented in Chapter II.

1.4.2 OOD Methodology. Once the definitions were established, a methodology was developed for OOD with the following objectives:

1. The methodology must provide a framework for implementing current object-oriented concepts.
2. The methodology must be independent of the paradigm used to state the systems requirements.
3. The methodology must be independent of the programming language to be used to implement the system.
4. The methodology must be able to adapt to new advances in object-oriented design concepts and practices.
5. The methodology must be useful for producing a complete design specification.
6. The methodology must be easy to use.

Validation of the methodology was accomplished through inspection, evaluation of a sample problem, and evaluation by a team of software engineering experts.

1.4.3 Requirements for a Decision Aid. Analysis of the design problem itself revealed the significance of the decisions involved in software design. That is, to produce a *good* design, *good* decisions must be made. Pressman's definition quoted earlier confirms this assessment. We concluded then, that an effective support environment must support the decision aspects of design. This led to the suggestion that the concepts of decision support systems (DSS) should be used to develop a prototype *Decision Aid for OOD.* Applicable DSS concepts are discussed in detail in Chapter II.

The methodologies for developing DSSs were used to determine the requirements for the prototype decision aid. The problem was concept mapped [48] and a

feature chart [41] and storyboards [5] were developed to represent the user decision processes embodied in the methodology.

The kernel system to implement as a prototype was chosen based on the concept maps of the requirements and design models, the storyboards representing the OOD methodology, and the capabilities of the environment chosen for implementation.

While a formal statement of requirements was produced, in keeping with the adaptive or iterative design philosophy [25], these requirements were never considered "final" or "frozen". Built into the DSS was a Hook Book [49] tool which was used to discover and record possible modifications, problems, and corrections—many of which were implemented, thus changing requirements. Chapter IV presents detailed requirements for the decision aid.

1.4.4 Implementation and Evaluation of the Prototype. Design of the decision aid began with the high level storyboards and feature chart and contined with the choice of components to be used to build the modelbase, database, and dialogue portions of the system. The specification for the dialogue component required that it be easily modifiable, able to support fast prototyping of a window and mouse based environment, and able to integrate with the existing environment. The Smalltalk/V Object Oriented Programming System (OOPS) was chosen to support this component and was inplemented on a Zenith Data Systems Z-248 micro-computer.

The DCDS [46] software development environment was used to support the initial requirements specification database. Hardware and software configuration problems prevented direct integration with the DCDS database, consequently the requirements were downloaded from the database to text and graphics files for access by the Smalltalk environment. The prototype's modelbase only consists of textual data for the heuristics and consistency checking aspects of the tool; so it was implemented via Smalltalk text windows. A detailed discussion of the design process and rationale for specific design decisions are presented in Chapter V.

Formal evaluation criteria for the Prototype was based on the Utilization-Oriented Evaluation model presented in [38]. This methodology is based on decisions which will be made regarding acceptance of the thesis, further research, etc. and applies specifically to the evaluation of DSS. Evaluation criteria and results are also presented in Chapter V.

1.5 Maximum Expected Gain

This thesis presents object-oriented design as a practical, and hopefully, usable design technique which extends, rather than requires obsolescence of current requirements, design, and programming methods and tools. The definitions and methods presented will aid a software developer's understanding of the object-oriented paradigm and its relationship to more familiar software development techniques.

The DSS techniques employed demonstrated their usefulness in developing software design support systems as well as application to other design environments. It is hoped this research will spur continued investigation of both OOD methodologies and DSS techniques by providing a test-bed for such work. Additionally, the tool may be used to support computer-aided instruction for OOD and related software engineering concepts.

II. Models and Concepts

2.1 Introduction

The purpose of this chapter is to describe the concepts and underlying models foundational to the thesis. We will first look at the object-oriented paradigm and how it has been applied to programming (OOP) and design (OOD). Next, a general model for an object-oriented design will be presented. A survey of requirements analysis and specification techniques follows along with a general model for stating requirements. Finally, we will describe the decision support techniques used to develop the decision aid.

Before getting into these concepts, we must describe the technique we used to understand and present them: *concept mapping.*

2.1.1 Concept Mapping. A concept map is a simple unstructured entity-relationship diagram with entities represented by ovals and relationships represented by directed arrows [34]. Following arrows from entity to entity provides a means for quickly identifying the relationships between those entities. Concept maps can describe one or more main entities, identified by directed arrows mostly pointing away from the entity. The concept maps in this thesis use a darker oval to represent central concepts for clairity.

The concept map is not only a means of representing an idea or stating a proposition, but can be used to elicit information from a knowledgeable source. Concept maps have been used by analysts, for example, in problem understanding and requirements determination for developing decision support systems [48].

The concept map in Figure 2.1 describes ideas about concept maps discussed by McFarren in [48]. The figures in this chapter, almost all concept maps, present an excellent insight into the material covered. That the information can be assimilated

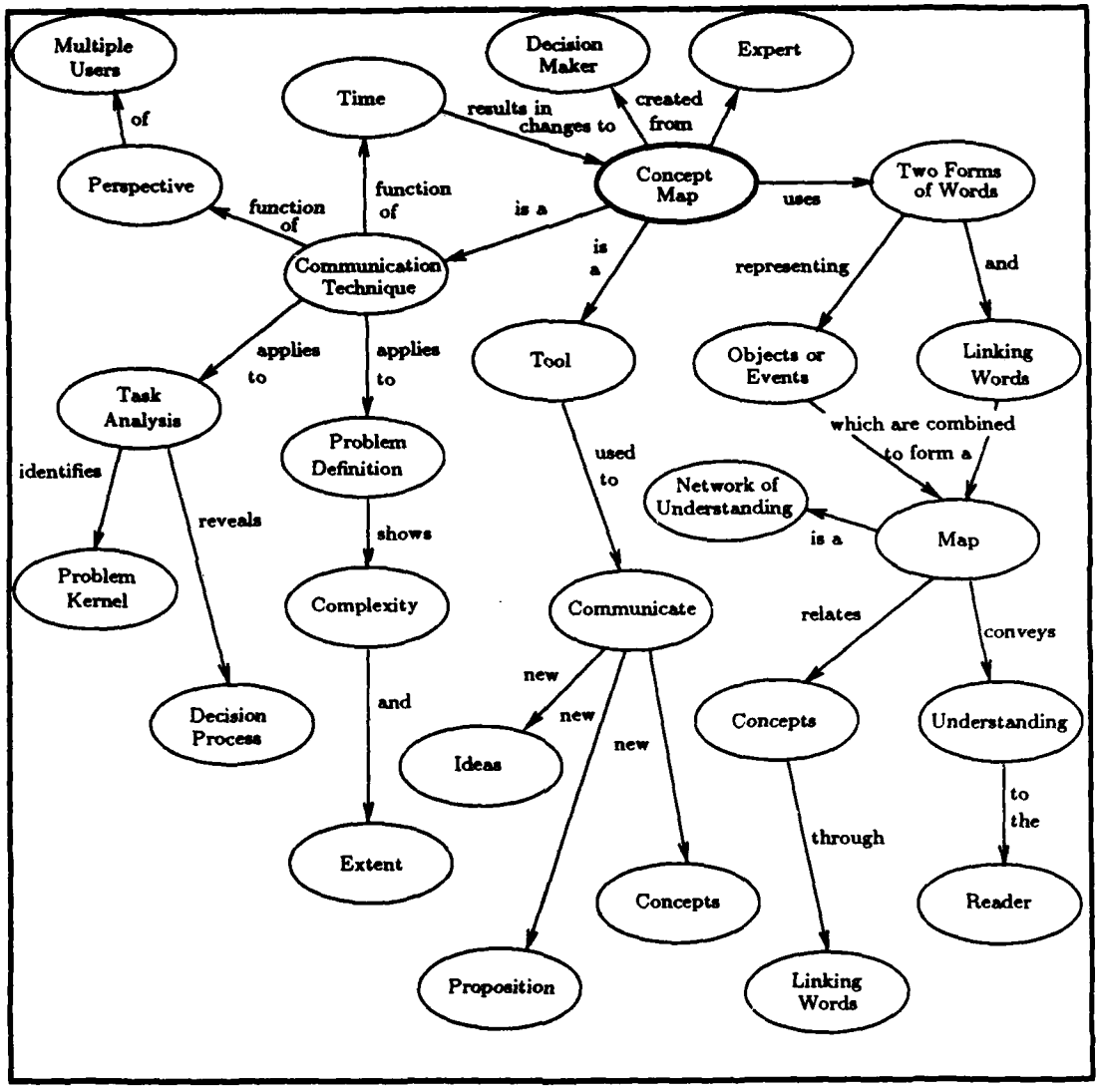


Figure 2.1. A Concept Map Describing "The Concept Map"

much quicker via the concept maps than by reading the text is a demonstration of the value of the tool.

2.2 The Object-Oriented Paradigm

The object-oriented paradigm was introduced briefly in the previous chapter. We said that it was a model for representing something in terms of its entities or objects, their attributes, and the operations which they may perform or which they suffer of other objects.

The object-oriented paradigm can be used to present a static view of the components which comprise a problem domain, as well as the behavior of such components under various conditions. A functional view, on the other hand, presents the steps or processes which must be performed to accomplish a single objective. In a mathematical sense, the object-oriented paradigm is analogous to *sets* while the functional paradigm is comparable to *algorithms*. Which representation to use would naturally depend on what we are trying to represent and why we want to represent it.

For four decades our computer software systems have been designed and programed functionally. That is, with some objective in mind, a series of steps is arrived at which accomplishes that objective, i.e., an algorithm. This method works fine for programs which accomplish a single function, or a sequence of related functions. But in the real world, events consist of complex interactions between entities performing possibly many functions simultaneously, each of which affects one another's state and subsequent behavior.

Functionally developed software systems that attempt to represent or control real world entities exhibit weak cohesion regarding the current state or behavior of those entities. State information is often scattered throughout global storage or exists only in temporary variables, which then become control flags. The presence of many such control flags promotes tight coupling. Procedural behaviors are of-

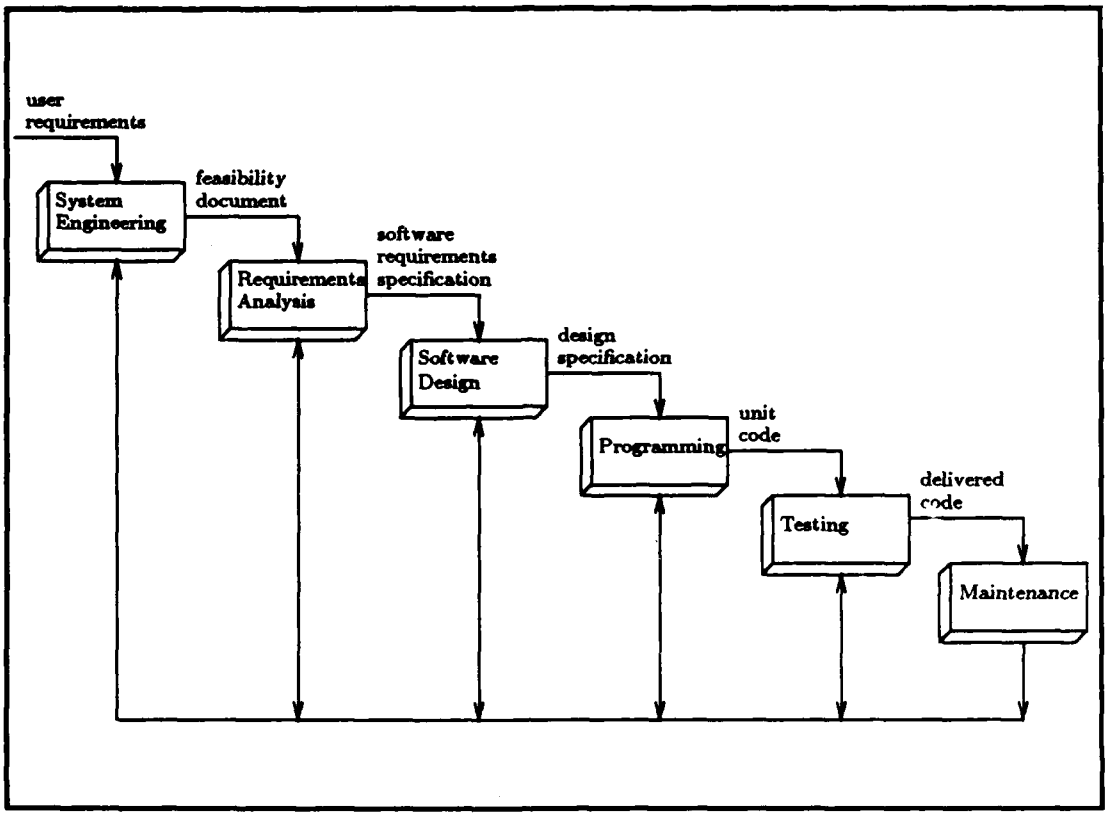


Figure 2.2. The Software Development Process

ten dependent on the current state of global data which may be modified by other seemingly unrelated procedures. These problems result in software which is unreliable and difficult to maintain. An object-oriented approach to software development shifts the emphasis from the functions to be performed to entities which must be represented.

In the following sections we will focus on life cycle implications of the paradigm with emphasis on programming and design.

2.2.1 The Object-Oriented Paradigm in the Life cycle. Figure 2.2 depicts the classic "waterfall model" of the software development life cycle as a multi-step process of translating user requirements into a computer language. What this implies is that

there is a *semantic gap* between the users' view of the problem and the computer's implementation of a solution. Each of the first several phases presents a new view or language level representation of the system and requires translation from the previous language. Many of the problems in software development stem from these translations. Because translating is a communicative process, it is fraught with ambiguity and misunderstanding [10].

It is this ambiguity software engineers would like to reduce or eliminate through use of the object-oriented paradigm. In this attempt, it has been applied to both design and programming, and researchers are currently exploring object-oriented requirements analysis techniques. A major concern is where the paradigm can be applied beneficially, and where it only muddies the water.

Figure 2.3 gives an overview of the application of object oriented concepts to design and programming. In addition, it shows the desirable software engineering concepts embodied in the paradigm. In his thesis examining the theoretical foundations for the paradigm, Bralick [10] states, "The object-oriented paradigm provides a natural structure for describing and decomposing systems." By using this more natural representation, the object-oriented paradigm lets us simplify translation by mapping language terminology more closely to that of the user.

Since the paradigm was first articulated in the context of programming, it's only natural that the first major research emphasis should be in developing object-oriented programming languages [13]. Because many object-oriented principles stem from this background, we will take a look at object-oriented programming before moving on to the design issue.

2.2.2 Object-Oriented Programming (OOP). In OOP an object is "some private data and a set of operations that can access that data" [13]. OOP systems consist primarily of many such objects communicating with one another via messages which invoke the target object's operations or methods. These messages act

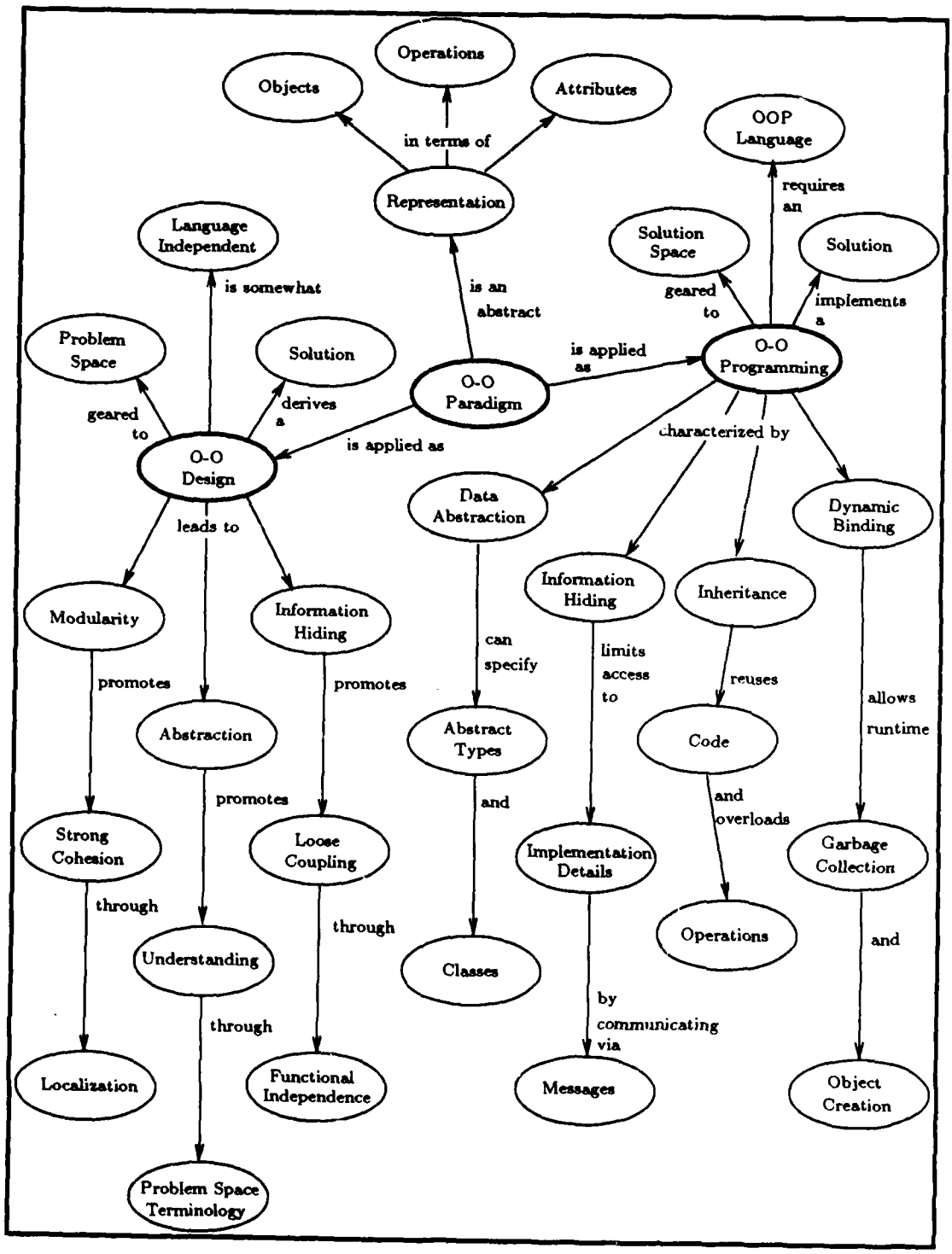


Figure 2.3. The Object-Oriented Paradigm

like mathematical functions, i.e. they return a value or the result of the operation and may require one or more parameters. In this sense, they are not very different from conventional function calls. In fact, an OOP method's detail code might look much like that of any modern programming language such as Pascal. We will soon see, however, that OOP languages possess or apply certain characteristics very differently than do conventional procedural languages.

It is generally agreed that to be object-oriented, a programming language must exhibit four characteristics: information hiding or encapsulation, data abstraction, dynamic binding, and inheritance of attributes and methods [36]. The implications of these principles to the capabilities of the language are what differentiates an OOP from conventional procedural languages.

Information hiding is more than the ability to modularize code. OOP requires the ability to represent objects, their state memory and operation code in a single module. Procedures in most languages cannot retain the state of local variables. Using global variables or passing pointers to external data only tightens the coupling between modules which is something we would like to avoid.

Data abstraction is the ability to represent and manipulate data structures in terms more analogous to their real world counterparts. More than just using meaningful labels for variables, abstraction requires the ability to define new types that relieve the programmer from having to know or mess with the underlying structure. Abstraction, as such, can be found in most modern procedural languages such as Pascal and Ada. However in an OOP language, the idea is carried a step further by requiring that an object only be manipulated through its operations [52].

Hybrid languages like Ada have been extended to support the structure of programs such that, at a high level, data structures may be private or hidden from other objects. However, in an OOP language such as Smalltalk, even the smallest number or character is an object manipulated through messages invoking its associated methods. Procedural languages explicitly define most operations through the

language's syntax rules, and implement them in the compiler's elaboration of the code. In an object-oriented language, even the simple operations of "+" and "–" are implemented as methods on objects such as integer or floating point and can be modified for new classes which might inherit such operations.

Thus while languages like Ada explicitly define constructs to support the representation of objects, the object-oriented paradigm is inherent in languages like Smalltalk. This is not to say one method is preferred over the other, but there is a clear distinction that must be understood in order to appreciate the concepts involved in OOP.

Dynamic binding requires that a data structure not be bound to a type or memory location or even size until program execution. This is just the opposite of the requirement for strong static binding found in most modern programming languages such as Ada, and involves a tradeoff between structure and flexibility. Its primary benefit lies in prototyping and simplifying experimentation when an object's exact structure or behavior may be initially unknown [52].

Inheritance and *class* refer to the static relationship between objects. The concept of objects belonging to some class allows us to categorize objects by their common structure and behavior as well as to create new classes by altering the characteristics of some existing class. Inheritance differs from object cloning or generic code instantiation in that we reuse unchanged behaviors by referencing rather than reproducing the code. It differs from use of library routines since operations are localized and automatically available to objects in the class without having to be explicitly included and called. Localizing code in a class structure and inheritance effectively reduces code bulk and simplifies program construction and debugging. However, it achieves this at the expense of increased overhead and tightened coupling of incrementally defined structures and operations within the classes.

We have highlighted some of the benefits and drawbacks of OOP languages and indicated why there exists such debate over which languages are object-oriented,

class-oriented, object-based, or excluded completely [52]. Our objective, however, is to see how OOP work relates to our main interest—design.

Cox states that his intention in using OOP is to try to avoid design altogether through reuse of code [27]. This would eliminate a level of language in development by using a programming language at such a high level that user requirements could be programmed directly. As language constructs become orders of magnitude more powerful, fewer are needed to build a system. Systems can thus be developed more quickly, are simpler and more reliable, require less maintenance, and programmers can become more productive. This goal is similar to the automatic programming approach which requires a language-based requirements specification which can be automatically translated into executable code [6].

The problem with both concepts is that historically, as components have become more powerful, rather than accepting simpler systems, users have wanted even more powerful and complex products. No one would suggest, for instance, that the advent of the 80186 microprocessor, which replaced about 20 individual chips, issued in an age of simple, automatically constructed computers. It merely provided room for more memory, co-processors, etc. Today's micros are just as crowded inside as yesterdays were; they just do a lot more. The same can be said for software. A few years ago, 64 kilobytes of user memory appeared to be an upper bound for microcomputer applications. Try to find a full featured word processor or spread sheet program for today's micro that will run in 64 K!

Thus, no matter how high-level we make the languages, there will always be larger and even more complex systems required whose efficiency and performance requirements dictate a rigorous software design be performed prior to programming. The application of OOP has therefore been toward *programming in the small*, and design has just not been a significant topic of discussion. Only within the last few years have we begun to see object-oriented design issues raised [27]. We will explore some of these issues in the next section.

2.2.3 Object-Oriented Design (OOD). Referring back to the waterfall model of Figure 2.2, we see that design is concerned with translating a requirements specification into a design specification. As we said in the introduction, both the "product" of design as well as the "process" must be addressed. We will first look at general attributes of the *products* involved in OOD, then explore the principles which apply to the *process*. A complete OOD methodology will be presented in Chapter III.

Any methodology for translation must take into account the languages we are translating from (the requirements specification) and to (the design specification). Figure 2.4 shows the basic attributes of these products.

In the general sense, all software requirements documents consist of at least a textual functional specification and an interface description. In addition, they are often complemented by graphical flow diagrams and/or a data dictionary. In the same way, software designs, regardless of methodology, must depict the modular architecture, the interfaces between modules, and the flow of control which describes the dynamic behavior of the system. A complete object-oriented design specification must represent all three of these attributes. We will describe the specifics of an OOD specification later in this chapter.

Knowing the components of the specifications, we can address the design process itself. The concept map of Figure 2.5 depicts various methods currently in use for carrying out the processes of analysis, design, and programming. From this diagram we can see the various underlying models or paradigms upon which the methods are based. Several questions are also brought to mind: "Is there a 'best' method for all problems?" "Are certain methods appropriate only to certain classes of problems?" "Should a single paradigm be used throughout the lifecycle?" We may not have definitive answers to these questions, but assuming that the object-oriented model is *good*, we hope to answer the question, "Can OOD be used without requiring object-oriented analysis and programming methods?"

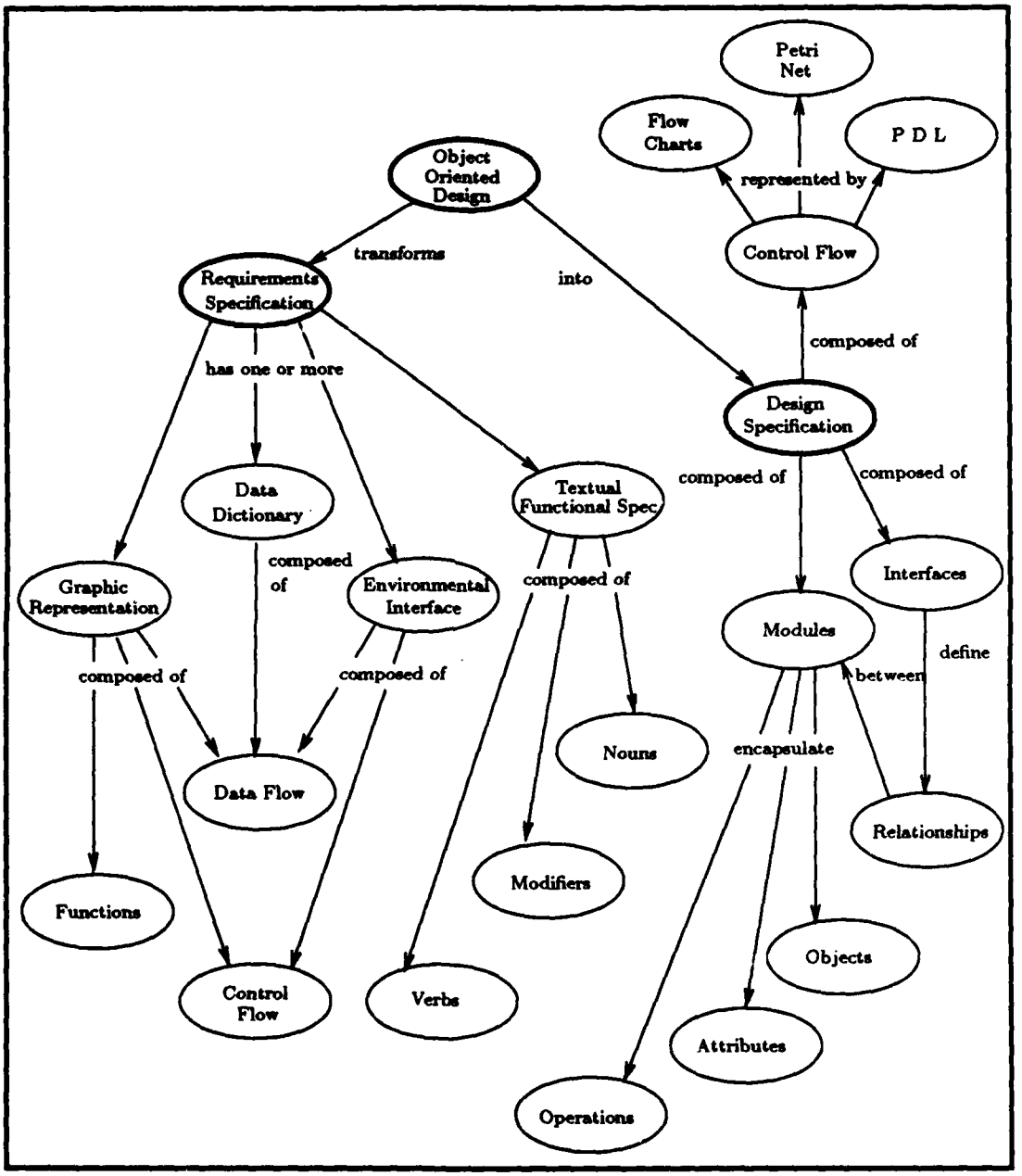


Figure 2.4. The Attributes of Software Development Specifications

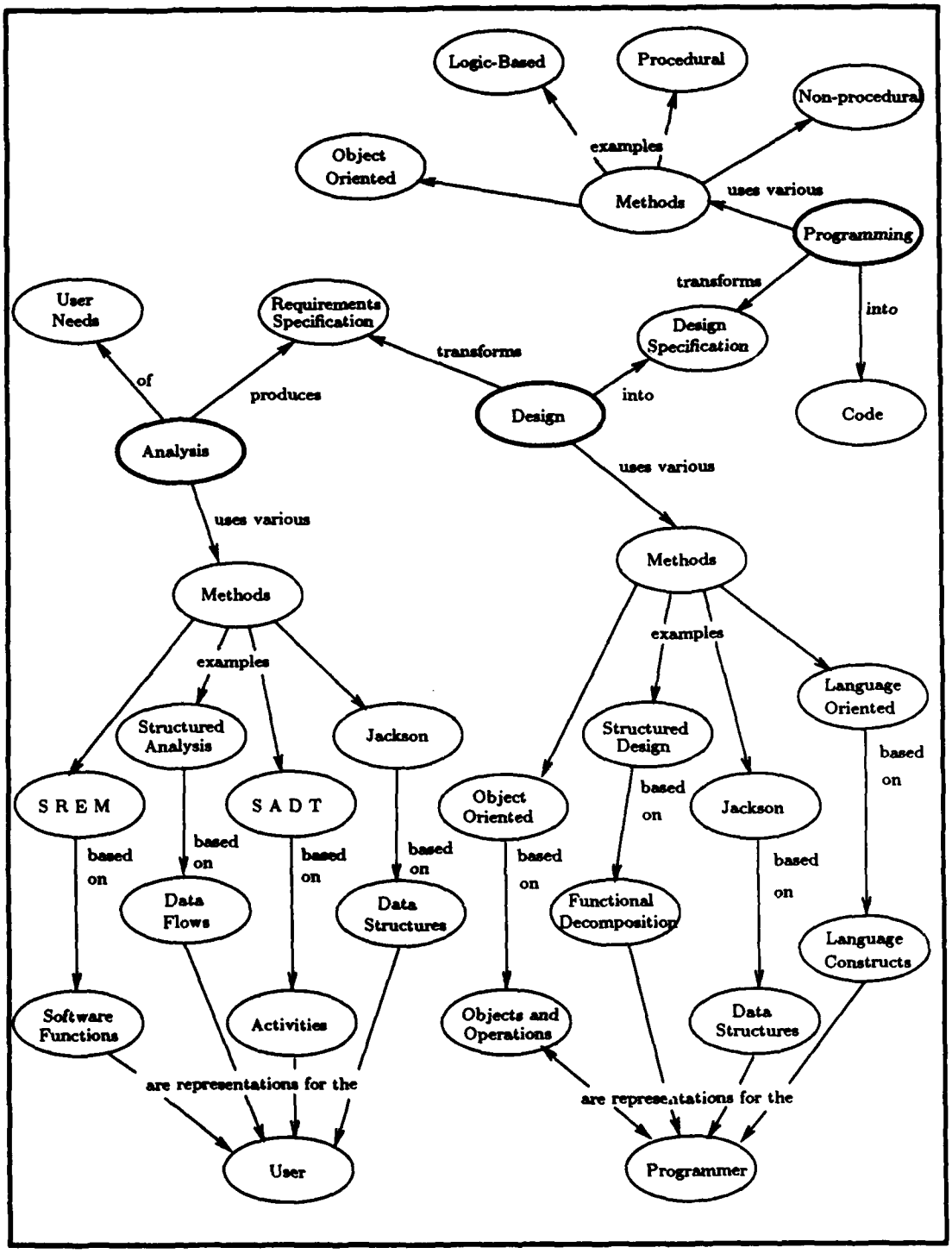


Figure 2.5. Software Development Methods

and rationale for specific design decisions are presented in Chapter V.

Just as there are general characteristics of specifications, there exist general software engineering principles which apply to the design process. Pressman [37] lists those specifically addressed by OOD as abstraction, information hiding, and modularity. While the principles themselves are not new, he states, "only OOD provides a mechanism that enables the designer to achieve all three without complexity or compromise." We have already seen how these principles can be compromised when certain classes of systems are developed functionally.

Comparing Pressman's list of principles with OOP characteristics reveals some parallels and some differences. This can be attributed to the distinction between the design and programming activities. Dynamic binding and inheritance are excluded from Pressman's list—presumably due to their strong implementation-orientation and applicability to experimental development methods which avoid a formal design phase. However, we believe inheritance may play an important role in design as well.

2.2.3.1 Abstraction. When describing a system, the first step is to simplify or narrow its scope so we can understand it at its highest "level of abstraction.". As we go into more detail, our abstraction changes. With OOD, our abstraction is in terms of its natural components—referred to and described in their own problem domain terminology, rather than as a set of processes needed to carry out the purpose or objective of the system.

Each component or *object* is characterized by its behavior and those assertions which may be made about its *state.* This state can change when the object suffers or performs some behavior. Objects, behaviors, and the attributes which represent types of assertions can be complex and thus composed of simpler sub-components.

Abstraction also lets us take a "black box" view of components of the system at their highest level. That is, we need not concern ourselves with low level details contained within the component. This component view enables creation of libraries

of reusable parts or sub-parts, reducing the number of levels of abstraction needed when decomposing the system.

2.2.3.2 Modularity. The concept of decomposing a system into simpler modules has been around for four decades [37]. The problem is not whether or not to modularize, but how to encapsulate system components into well structured modules.

Traditionally, modules were determined by the functional decomposition of the system, i.e., mapping functional requirements directly to the partitioning of the design. In OOD, however, a module is identified as a data structure and the operations that act on it. Both the data structure and its operations are co-located in that module or object. This gives OOD modules strong cohesion.

Cohesion is strongest when modules perform a single function [37]. In procedural designs, cohesion may be strong at the lowest level, but for large problems, hierarchical structuring requires combining activities such that at higher levels cohesion is weak. These high level functions are often described in such arbitrary and ambiguous terms as "Process_Message". With OOD we say a module has strong cohesion if it represents a single entity. Even at the highest level of a system composed of a set of communicating objects, the system itself can be considered a single object. A complex top level object such as "Communications_System" is neither arbitrary nor ambiguous and describes a very specific problem domain entity.

2.2.3.3 Information Hiding. Not only can details be hidden by levels of abstraction, they can also be hidden and protected from other components. In OOD, a component cannot access another's private data structure, only its visible interface. This promotes the principle of "loose coupling" in that it eliminates hidden dependencies between modules. Information hiding carries implications for development and maintenance, as well as reliability of operation.

Once a module's interface has been determined, lower level design decisions are isolated to a specific component of the system and do not affect development of other components. This characteristic, called "functional independence," makes object behaviors straight forward, simplifies testing, and makes systems more reliable. Since the effects of a change are localized, debugging and maintenance are also streamlined.

2.2.3.4 Inheritance. In OOD, objects are determined from the problem domain rather than a solution-space class hierarchy. Operations and attributes aren't inherited, they are *observed* through analysis of the object's discernable behavior within the scope of the problem. As such, inheritance doesn't seem to apply in the analysis, identification, and encapsulation of objects. However, once the specification for an object is developed at its highest level, inheritance may be used in the decomposition, or rather construction of the object. Seidewitz points out that inheritance should be hidden in design through this type of *bottom-up* application [27].

We are speaking of the reuse of concepts or frameworks here rather than the reuse of code. That is, if we recognize in an object a familiar structure and behavior, we need only specify that this *new* object is an instance of some *known* class of objects with possibly some minor modifications. The decision to refer to the known class is based on the significance of the differences attributed to the new object, i.e., the cost/benefit of modification versus redesign.

While Pressman offers an example of design using inheritance involving geometric shapes [37], many real system involve interacting dissimilar objects which do not fit into such a neat class hierarchy. The author's experience indicates that inheritance in design is seldom formally used since supporting resources are not yet widely available. Formalization would require a library of generic software module designs which would need to be maintained, cataloged, and readily available in the designer's environment. Informally however, a software engineer often applies the

essence of inheritance by recalling a previous design, or retrieving one from some previous work, and massaging it to fit the new problem. This type of reuse improves productivity and should be supported in any modern design environment.

2.3 The Object Model

In this section we will formally define the underlying object model used for development of the thesis and describe its representation in a design specification.

2.3.1 Definition of the Object Model. All along we have been using the terms "object", "operation", and "attribute" to describe components of the object model. We have also mentioned the additional terms "method" and "message". We have referred to several informal definitions used for these terms based on the author and context of the discussion. But since definitions vary slightly from author to author, it is necessary to establish a working definition of the object model for our purposes.

We begin with two definitions of the object model: Bralick's theoretical model [10] and the Smalltalk model [20], then proceed to define a model appropriate to design. We chose Bralick's model because it seemed to be the only attempt to provide a rigorous, firmly founded definition of the paradigm. The Smalltalk model, on the other hand, seems to be the defacto standard to which all other proposed implementations of the paradigm are measured.

2.3.1.1 A Theoretical Object Model. Bralick's object model is represented in Figure 2.6 and the components of the model are defined as follows:

object A unique entity composed of an *identifier*, a set of *attributes*, a set of *behaviors*, and a set of *objects*, and having a *link* to its parent *object*.

attribute A property of an entity or *object* which associates a value from a domain of values with the entity at a point in time. A such, it serves to limit, identify, or describe an object. An *attribute* is composed of an *identifier*, a *value*, and a set of *attributes*, and has a *link* to its parent *attribute*.

behavior An action an *object* can perform which results in a change in the state of the *attribute(s)* of some object. A *behavior* is composed of an *identifier*, a set of *attributes*, and a set of *behaviors*, has a *link* to its parent behavior, and may map to an *operation*.

identifier An arbitrary string which uniquely identifies an *object*, *behavior* or *attribute* within some context.

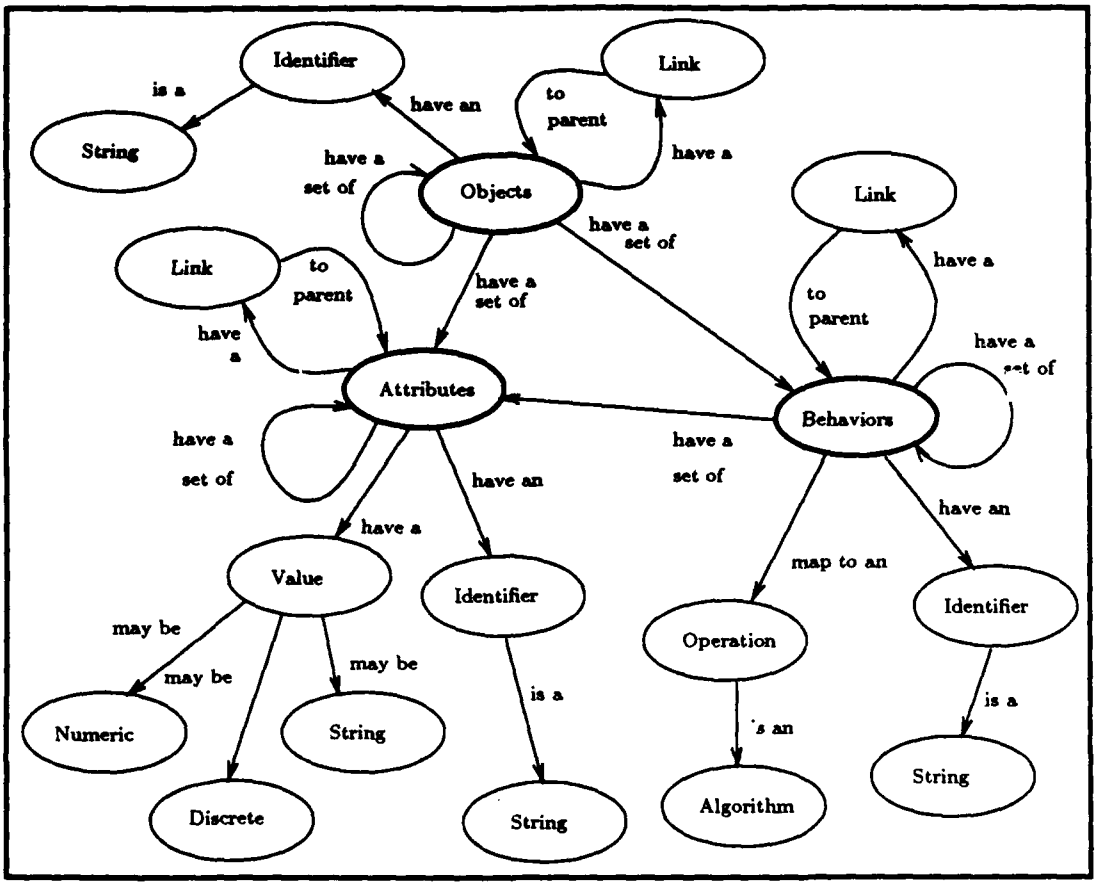


Figure 2.6. Bralick's Theoretical Object Model [10]

value Represented as either a numeric value, a selection from a discrete set of possible values, or an arbitrary string.

operation The algorithmic description or sequence of executable statements performable on some object.

link An association of an entity with the parent or owner of the set to which the entity belongs.

This definition reveals the fully recursive nature of objects. Objects can be constructed of, or decomposed into sub-objects; behaviors can invoke one or more sub-behaviors; and attributes can be described generally, then gradually broken down into increasingly complex data structures. The model also supports relationships

Table 2.1. Object Classification [10]

Object Type	*Affects*		*Affected By*
	Others	*Self*	*Others*
Static	N	N	N
Passive	N	N	Y
Small	N	Y	N
Weak	N	Y	Y
Demon	Y	N	N
Interactive	Y	N	Y
Sovereign	Y	Y	N
Complex	Y	Y	Y

between objects through their attributes. That is, since an attribute's value can be a string, it can act as a reference to another object's identifier.

Bralick also demonstrated the paradigm's ability to represent any computable function, i.e., it is at least as powerful as a Turing Machine. This is what we need in design: a medium powerful enough to represent any problem in its own natural terms, rather than squeezing it into the restrictions or terminology of a programming language or methodology.

Rather than limit an object to only representing a state machine, Bralick lists eight general classifications of objects based on the nature of their behaviors. These types are listed in Table 2.1. This might be considered an extension of Booch's classification of objects as actors, agents, and servers, and of operations as constructors, selectors, and iterators [8]. Such classification schemes can be very useful in determining how to best associate objects with the appropriate operations.

Attempting to use the model for design reveals two significant limitations. First, is a lack of an explicit means for describing object interactions. Objects exist, they behave, they have state, they can be complex, but they have no means of executing the behaviors they require or suffer of other objects.

Second, the model suffers from being too ambiguous for use in design. For instance, a single data structure may be described variously as an object with a set of attributes, a single attribute which itself contains a subset of attributes, a set of sub-objects which each have attributes or sub-objects, etc. Such flexibility does not provide a designer with a clear picture of how to proceed in defining an object. Although this ambiguity may be useful for some purposes[1], precise communication is the objective of design.

2.3.1.2 An Object Model for Programming. The Smalltalk model, shown in Figure 2.7, is not so clearly spelled out or defended theoretically as Bralick's model; however, it does effectively accomplish its intended purpose—implementing an object-oriented language. This is of significance to us since we require a model capable of representing the real world complex interrelationships between objects. From [20] and [17] we present the following definition:

object A self-describing, protected data structure which encapsulates information and provides functionality. Every *object* is an instance of some *class.*

class A program module which defines the behavior of similar *objects* by specifying the *variables* they contain and the *methods* available for responding to *messages* sent to them. *Classes* are also *objects* contained in *global variables* so they can be referred to in *expressions. Classes* are arranged in a hierarchy where each *class* is a *subclass* of some other *class* or the root *class* called *Object.*

subclass A *class* which inherits the functionality of all its *superclasses* in the hierarchy. Each class builds on its *superclasses* by adding its own *methods* and *variables.*

variable A container for a single *object* which can be of three kinds: *instance*, *temporary* or *shared.*

instance variables Represent the internal state or private memory of an *object* and may be referred to by name or by an integer index. Each member of a *class* has its own separate *instance variables* which exist for the life of the *object.*

temporary variable Created in and exists for the lifetime of a *method*; and act as *method arguments*, *method temporaries*, or *block arguments.*

[1] Bralick cites human communication as being ambiguous as an example of the value of flexibility in the model [10]

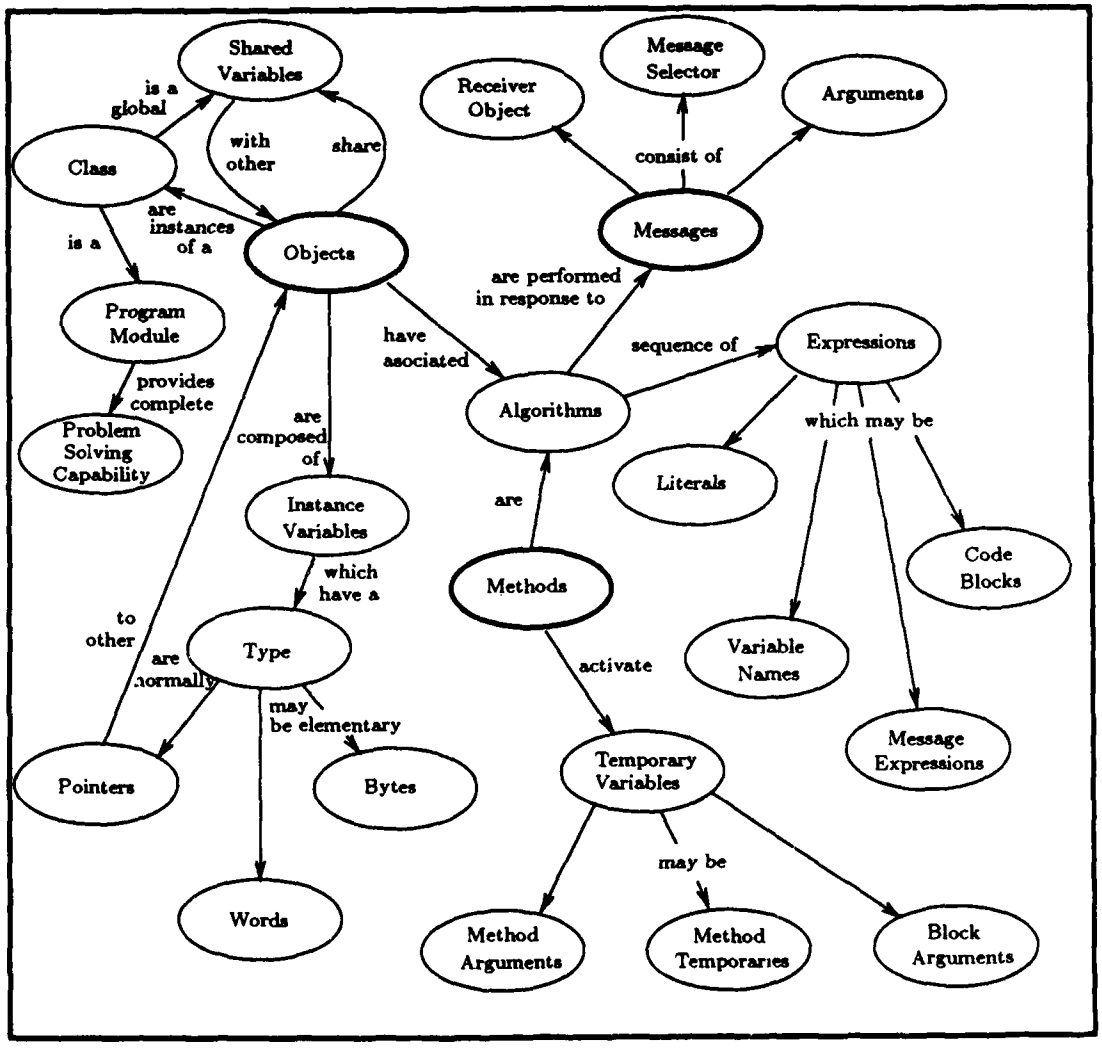


Figure 2.7. The Smalltalk Object Model [17]

shared variable *Objects* defined in dictionaries or pools which are accessible by other *objects*. *Global variables* are available from every *object* while *class variables* are only available to the *class*, *subclasses*, and instances of the *class* and its *subclasses*.

method An algorithm performed by an *object* in response to receiving a *message*. A *method* may change that *object's* state or send *messages* to other *objects*. *Class methods* implement *messages* sent to the *class*, while *instance methods* implement *messages* sent to instances of the *class*.

message A request for an *object* to perform an operation. Identified by a *message selector*, it specifies what to do, but not how an operation should be performed. The *receiver* of a *message* is either the *class object* or an *object* that is and instance of the *class* that defines the *method*.

From the definition, we can immediately see the implementation and language specific features of the model. We can also see most of the elements of the theoretical model. For example, the instance variables represent both an object's set of sub-objects and its set of attributes, which, since they themselves represent objects, provide for a recursive decomposition. An explicit decomposition of methods into sub-methods is not given. However, since methods may send messages to their own or other objects, invoking other methods, complex behaviors can be described.

Through messages, Smalltalk provides a means of representing the interrelationship between objects. While Bralick addresses this subject in his thesis, he makes no explicit provision for this interaction in the model. The problem with the concept of messages is that it limits object interaction to a single kind, which may not clearly or easily represent complex interrelationships. Bralick [10] provides an example the kind of convoluted thinking forced by the message model. He describes the process of drilling a hole in a piece of metal as follows:

> We are left with the counter-intuitive model of, for example, a piece of sheet metal being asked by a drill press to please punch a hole in itself. The sheet metal then decides whether to honor the request. Note that whether a hole is actually made in a piece of sheet metal by a given drill press is a complex interrelationship among the material of the sheet metal, the material of the drill, the speed at which the drill bit is rotating,

the force at which the drill descends onto the sheet metal, and how long the drill is applied in such a fashion to the sheet metal [10].

2.3.1.3 An Object Model for Design. Both of the previously discussed models leave something to be desired for application to design. The theoretical model is too ambiguous and the OOP model to restrictive to implementation constructs. The theoretical model provides no means for describing object interaction, and even Smalltalk's message syntax is rather limited. We proceed now to develop an object model which meets the needs of *design.*

We begin with the simpler theoretical model, and in the OOP tradition, modify it to meet our needs. First, we generalize the notion of an attribute. Since an attribute is "an object closely associated with or belonging to a specific person, thing, or office," [51], we will use attributes as a means of associating objects with other objects, other attributes, and operations. This is analogous to the Smalltalk variable being a pointer to some object.

Thus as we speak of what an object has or does, we are speaking of its attributes. Some attributes are important enough to be considered *required.* When we refer to such attributes as being required, we simply mean they must be accounted for. In some cases, a required attribute may be null, but it is important for the designer, and later implementers to know that the attribute is null and for what reason.

We may eliminate the explicit notion of an object having a set of sub-objects since such a set may be referred to by one or more of the object's attributes. Typical attributes representing object sets might be *component_objects*, *actor_objects*, or *server_objects.* An object's *name*, *class*, *parent*, and set of *operations* are all attributes of the object which may, in some cases, be null.

We retain the notion that an attribute can represent a set, but we expand that set to represent objects, other attributes, or operations. Attributes serve to identify

an object—by its name, class, behavior, and domain; and to associate an object with other objects, attributes, and operations. We will refer to attributes which relate objects or operations to other objects and operations as *relations.*

We reserve the term "behavior" to define a general description of what an object does in its response to stimulus from other objects, and use "operation" to tell how it performs that behavior. Thus an object's behavior would be an informal description of the object's functionality at the highest level of abstraction, while its operations would be a formally specified set of algorithms. An operation requires a means of invocation and an interface description. Therefore, we allow an operation to have attributes, like objects, which associate the operation with the sets of objects it modifies or requires as arguments, as well as those for which it requires services and performs services.

Bralick referred to object "cloning" in his thesis as a type of inheritance. This technique also more closely fits the Ada generic construct. We use the concept of *class* or *template* from which an object may inherit properties such as required attributes. Which method of class or generic implementation is used would depend on the programming language of implementation.

The model we have just informally described is pictured in Figure 2.8 and more formally defined as follows:

object A unique entity defined by *attributes* which serve to identify the *object* and *relations* which associate it with other *objects*, *relations*, and *operations*. Required *attributes* are *name*, *behavior*, *domain*, and *class*. Relations which may or may not be null include sets of *operations*, *component objects*, *actor objects*, and *server objects*.

attribute Serves to identify an *object* or *operation*.

relation Represents an association of an *object* or *operation* with other system *objects* and *operations*.

operation Is the description of how an *object* performs some behavior. Required *attributes* are *name* and *algorithm*. Relations include sets of *actor operations*, *server operations*, *argument objects*, and *modified objects*.

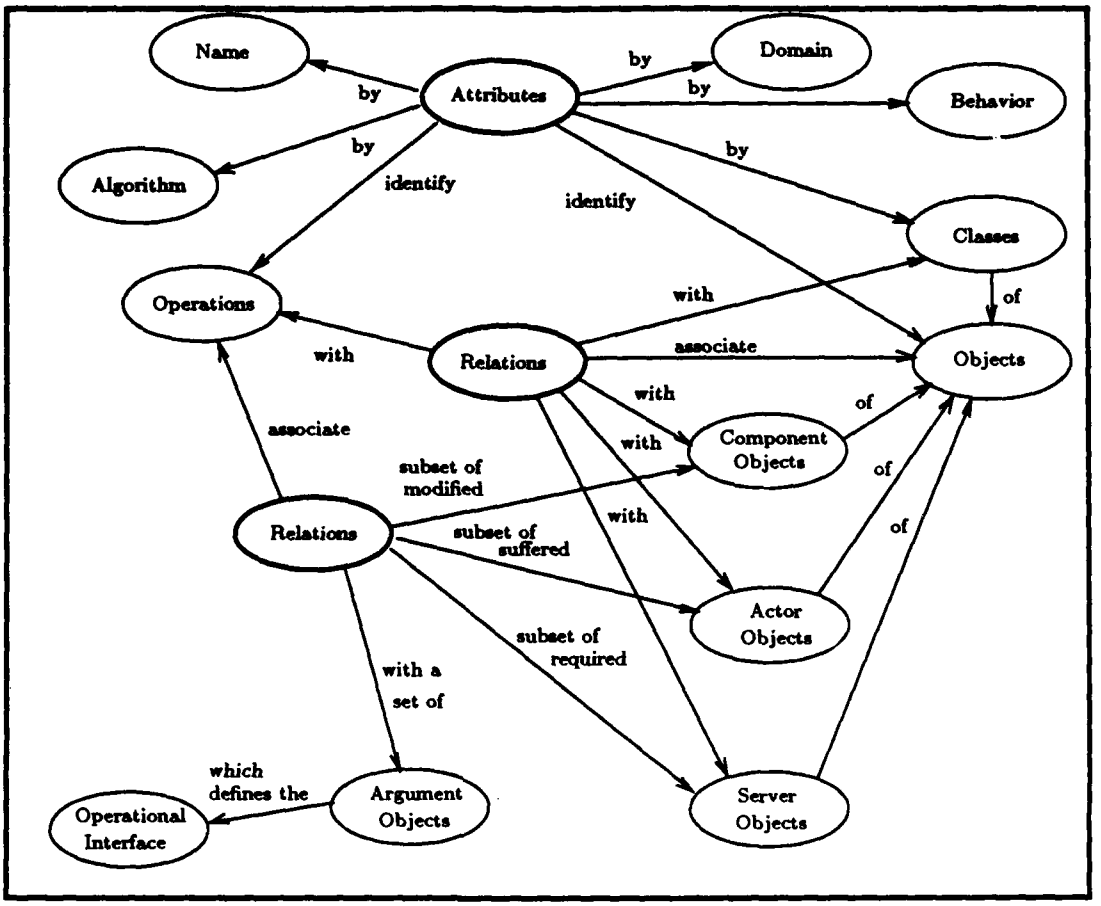


Figure 2.8. An Object Model for Design

class A complete design of an *object* which may be used as a template from which an *object* derives its characteristic structure and function.

name A string serving to identify an *object* or *operation* which must be unique within a context.

behavior A text description of an *object's* function when provided with certain circumstances.

domain A text description of the set of states to which an *object* may change.

actors A relation which denotes which *objects* or *operations* require services of some other *object* and *operation.*

servers A relation which denotes which *objects* or *operations* provide services to some other *object* and *operation.*

components A relation which denotes which *objects* can be considered logical parts of an *object.*

arguments A relation which denotes which *objects* are required as arguments in the interface of an *operation.* This relation has an attribute: *mode* which may be input or output.

modifies A relation which denotes which *objects* are modified by the execution of an *operation.*

We submit that the model presented retains the function of the theoretical model, and adds the practicality of the programming model, without suffering the limitations of either. Neither the implementation of an object is specified, nor is the syntax of the communication between objects limited to a specific method. Yet provisions are made for describing the interface between objects and operations of other objects, as well as for representing the fully recursive nature of real world objects.

2.3.2 Representing The Object Model. Statically, an object-oriented design consists of a representation of a system in terms of the model described in the previous section. As such, the object model could be easily represented in a relational database. However, a static representation is insufficient for fully communicating a complex behavior or the interrelationship between objects without a correspondingly complex textual narrative.

As an alternative to text, software developers have produced a plethora of graphical methods of representing software systems. A number of techniques have been proposed to represent an object-oriented design, some entirely new, some variations on more familiar methods. We will look at some of these in the following section, then offer one of our own.

2.3.2.1 Some Graphical OOD Methodologies. Examples of graphical OOD methods are shown in Figures 2.9–2.12. Each of the many methods which

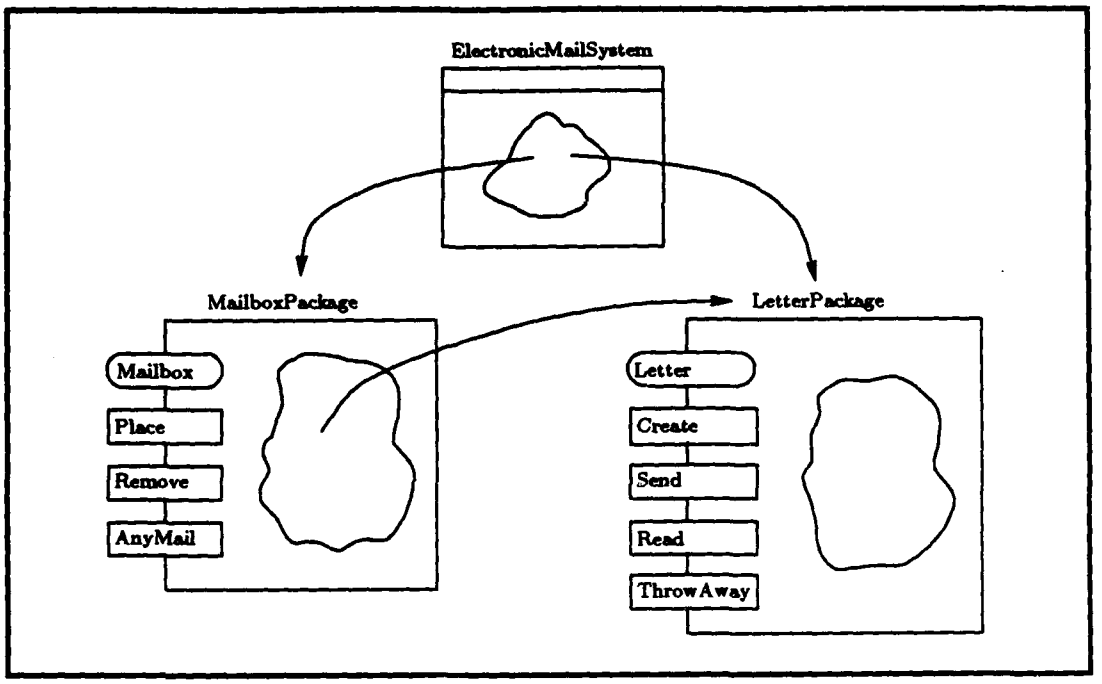


Figure 2.9. Booch Diagram Example [19]

have been developed has its own strengths and weaknesses and represent one or more views of the software design.

The Booch diagram [9] [19] of Figure 2.9 identifies the objects and operations in the visible interface, and the dependencies between objects, but it does not reveal which objects invoke which operations. Thus the diagram is useful only as a general block or overview diagram. Furthermore, attempting to show decompositions becomes immediately difficult.

Figure 2.10 is an example object diagram of Goddard Space Flight Center's General Object-Oriented Development methodology [42]. A variation on structure charts [35], GSFC's object diagrams are even simpler block diagrams. But they add the capability to show a clean parent-child or a virtual machine hierarchy. At the lowest level, object diagrams include the procedures and data stores, making them virtually indistinguishable from structure charts.

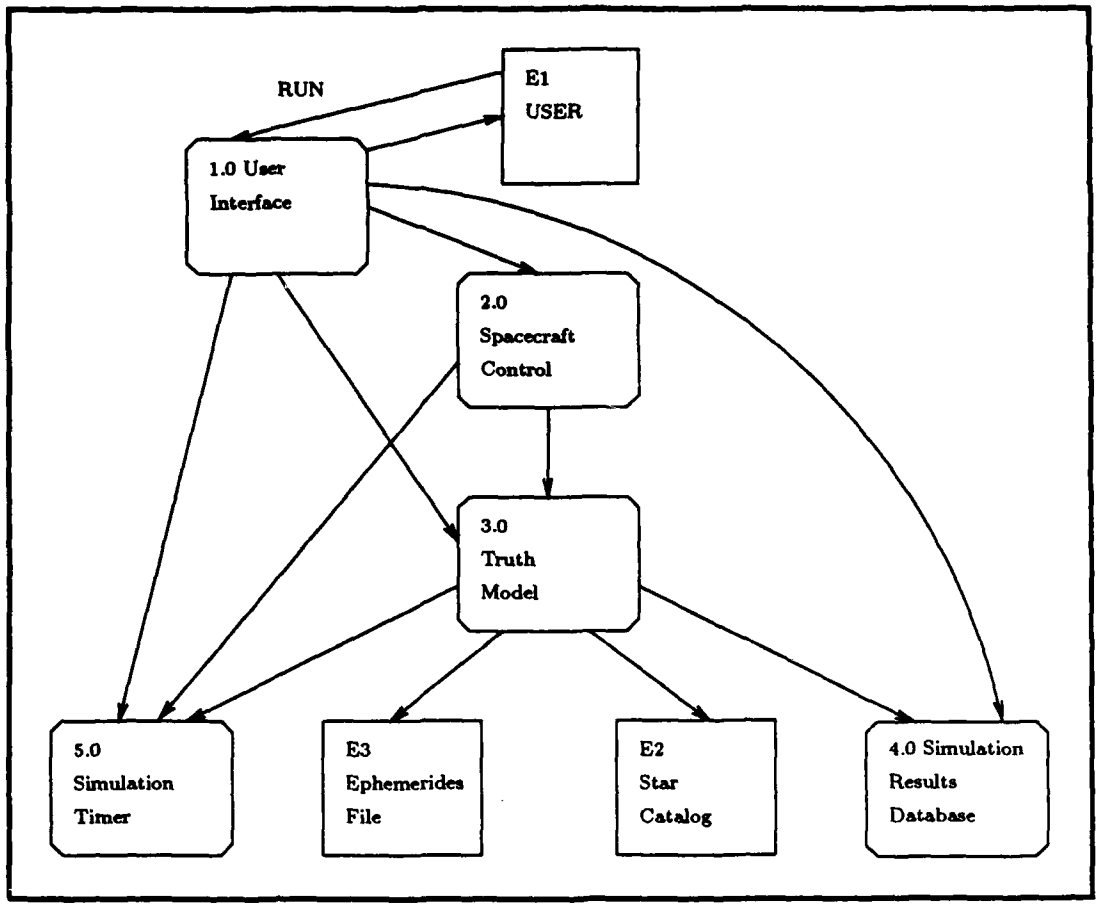


Figure 2.10. GSFC's Object Diagram Example [42]

Modular design charts [53] and Buhr diagrams [11] go into much greater detail. The modular design chart, Figure 2.11 shows attribute types and operations within an object, as well as which components are used by specific object bodies. The Buhr diagrams link operations together directly through "control sockets" giving the flavor of a hardware wiring diagram.

The Interactive Ada Workstation (IAW) [22] implements Buhr diagrams and adds a petri net diagram for describing control flow. The AdaGraph tool [14] which implements Cherry's PAMELA methodology also uses a petri net based process

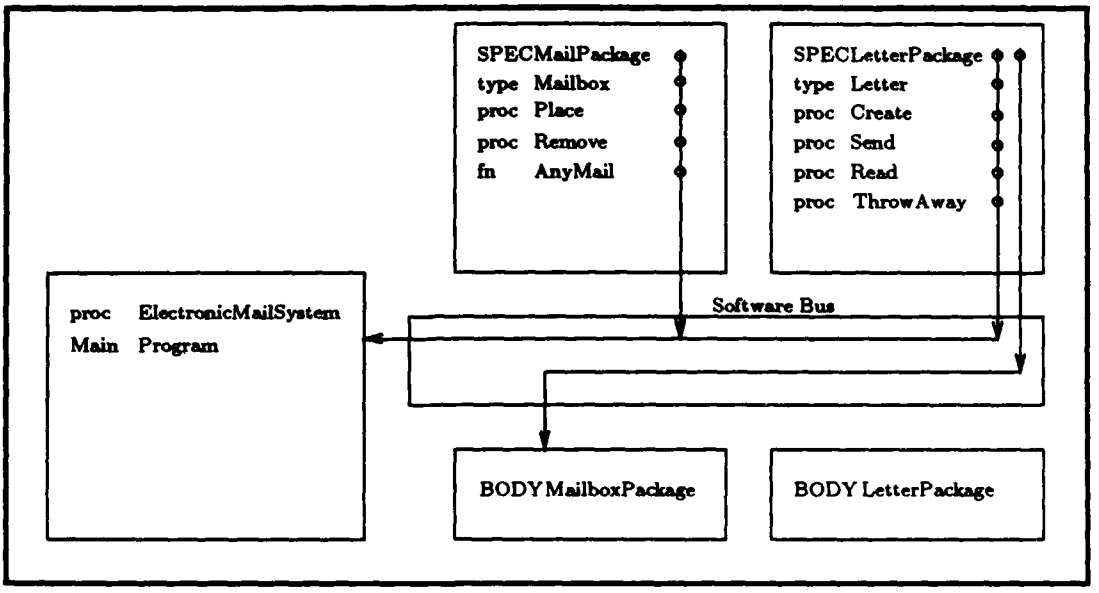


Figure 2.11. Modular Design Chart Example [53]

graph and adds a hierarchical subprogram graph. Both of these systems generate skeleton Ada code.

APEX, a system in development at the Air Force Wright Aeronautical Laboratories, also adds a petri-net diagram to its block diagram and process connection graph [2]. An example is shown in Figure 2.12. This system, like AdaGraph and the IAW, automatically produces an Ada shell.

The SHARP methodology [12] uses a variety of pictographs employing icons which seem to be an extension of Booch diagrams. Different diagrams are used for main program abstraction, object implementations, object interactions, object invocation, task rendezvous, subprogram data flow, data structures, and program unit operations.

All the methodologies we have referenced were developed specifically for designing Ada programs[2], resulting in many Ada unique distinctions. This makes sense

[2]The modular design charts were developed with both Ada and Modula2 in mind.

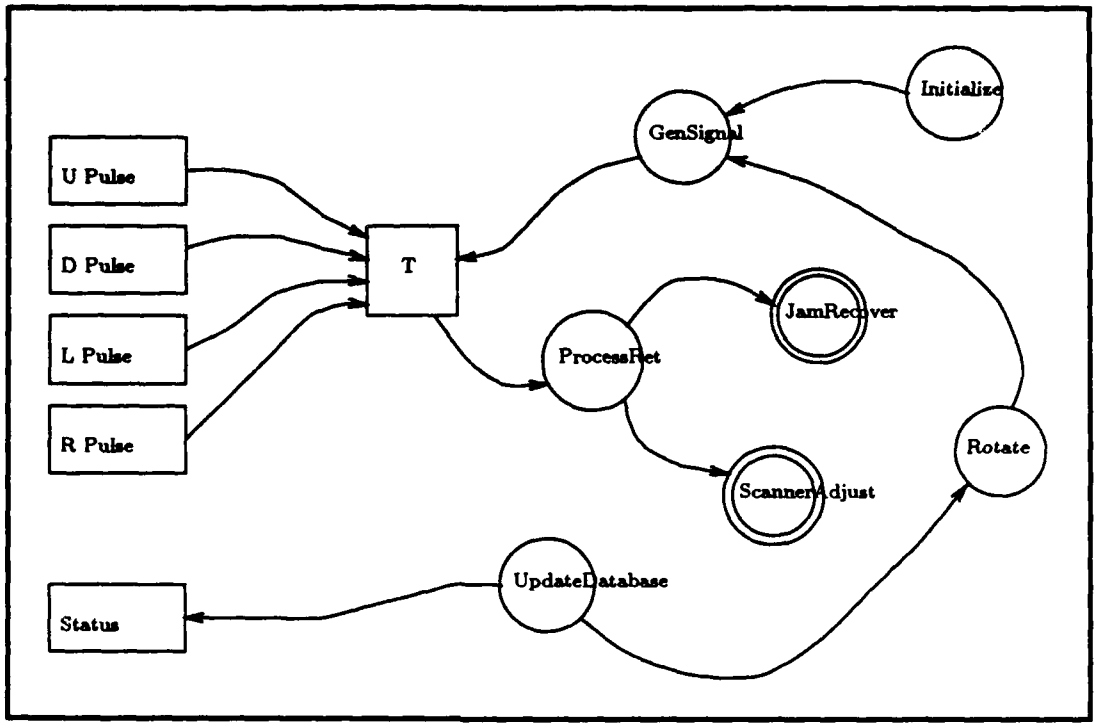


Figure 2.12. APEX Petri Net Graph Example [2]

if one takes the view that a design methodology must support language features and produce source code—as do the APEX developers [2]. It can also be attributed to the current emphasis on Ada by the DoD and the need to take advantage of its object-oriented features. Be that as it may, as more use is made of object-oriented languages other than Ada, so the need for a generic methodology which can be mapped into any such language is becoming more apparent.

2.3.2.2 A Graphical Representation For Generic OOD. Synthesis of the various types of diagrams used by the methodologies described previously yields an interesting parallel to electronic circuit design. With hardware design, both block diagrams and detail diagrams are important. Usually a block diagram showing the static relationship between board components is given, as well as a wiring diagram revealing exact pin connections. In addition, timing diagrams are often used to de-

scribe the dynamic behavior of the system by depicting the relationship between the signals being passed throughout the system. Could it be that electrical engineers have been using OOD for years and we are only just now catching on in applying it to software?

Since OOD seems to closely parallel the hardware design methodology just described, it would seem that a means of representing that design should also parallel the block, circuit, and timing diagrams used in the hardware design. We submit that this logic holds and that an OOD representation must consist of three parts: a *block diagram*, an *interface diagram*, and a *control flow diagram*. Figure 2.13 shows an example of a proposed design including these three views.

The block diagram we use is similar to the high level object diagram of Figure 2.10. It depicts the objects in the system (at a particular level of detail) and the dependency relationships between them. A module dependency is shown by directed arrows to the servant or component objects in the graph. In the case of an actor/server relationship, messages or operation calls flow accross the directed arrows.

The detail diagram is taken from the modular design chart shown in Figure 2.11. We leave out the constraints of depicting a "software bus" and component bodies as separate from their specification parts. In lieu of the implementation-oriented terms "package", "proc", "fn", and "type", we let objects begin with a capital letter, and operations begin with lower case.

We also agree with using an optional petri net graph to depict a state diagram or object interaction in the case of concurrent communicating objects. However, we would not want to require such a representation where it is not needed.

The main purpose of graphics is to communicate the design *more clearly than does the text.* While we advocate the use of graphics, we do not advocate a methodology so rigid that the graphic techniques drive the design, rather than good software

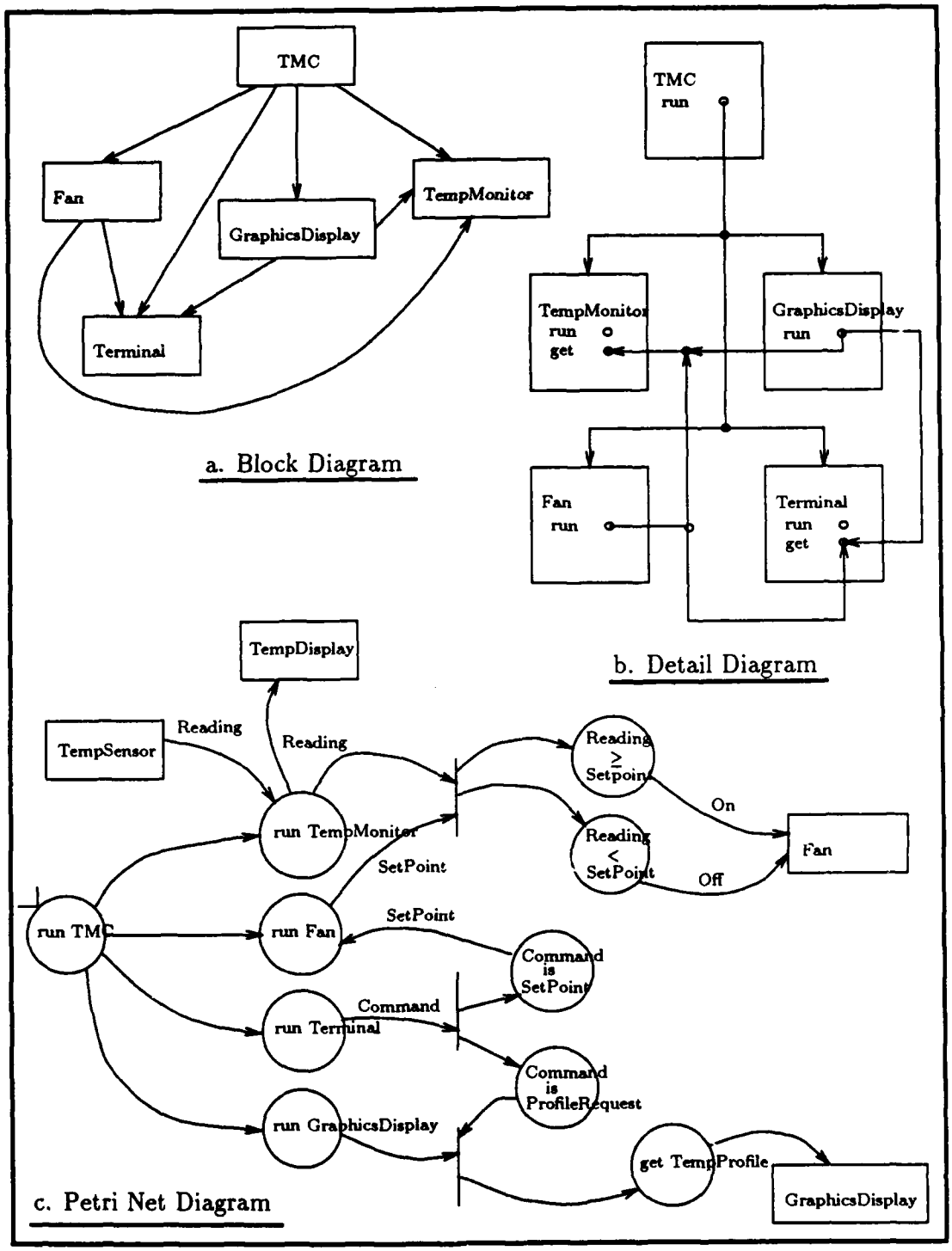


Figure 2.13. A Language Independent Object-Oriented Design

engineering principles. We feel a three view approach to a graphical representation is sufficient to effectively communicate the design.

2.4 Requirements Analysis and Specification Techniques

Referring back to Figure 2.5 we showed analysis techniques based on functional, data-flow-oriented, and data-structure-oriented paradigms. While some experts feel it may be too difficult to get an OOD from a specification not developed with the object-oriented paradigm in mind [27], Figure 2.4 indicates there may be some form of mapping from any of these specifications to OOD.

Whatever the paradigm behind the analysis, requirements specifications all contain some sort of functional text description, written in English or an English-like language. From the nouns, verbs, and modifiers in the text, objects, operations, and attributes can be determined using Abbot's semantical analysis methods [1]. Examples of such textual descriptions are the mini-specs of structured analysis and facing page texts of SADT.

Another common element of many methodologies is the data dictionary. While varying in format, these contain data items representing potential objects and attributes. Data flow and activity diagrams add the processes and interface descriptions needed to identify operations and visibility requirements. Control flows shown in SADT diagrams often represent design decisions rather than requirements and can be used to classify objects and determine object dependencies.

Finally, since most specifications are hierarchically organized, scoping the problem so a context can be determined is usually straight-forward. Since detail design uses conventional methods, low level processing requirements may map directly to low level operations. As simple as this all sounds, there is no absolute or magic in such a mapping. Each selection of an object, operation, or encapsulation requires application of the object-oriented and software engineering principles we have been discussing throughout this document.

Attempts have been made to provide formal translation techniques from requirements specification methods to ood [3] [42]. However, the formality of such methods severly limits their usefulness since one is constrained to specific specification formats. In addition, we found the cited methodologies significantly more complex than other more general design methods.

2.5 *The Requirements Model*

We define a requirements specification as providing a textual functional description of the system requirements, a data dictionary of required system entities, and an interface description depicting the flow of control and/or data through the system in operation and in conjunction with any external systems. These three components map well to those produced by common specification techniques such as Structured Analysis [15] and SADT [40].

However, this general model does not depend on a specific graphic representation or specification format. Our design methodology, described in the next chapter, requires only that this information be available to the designer in printed or automated form. Since we are presenting the methodology in a computer-based interactive form, we have chosen a requirements methodology supported by an automated tool: the Software Requirements Engineering Methodology (SREM) and the DCDS Support System.

2.5.1 *The Distributed Computing Design System* [46].

The DCDS is not just a software requirements methodology but is a unified environment for systems development. It includes methodologies for developing system requirements (SYSREM), software requirements (SREM), distributed top-level design (DDM), algorithms and unit code (MDM), and complete integrated system testing (TSM). Each methodology has its own customized language based on an element-attribute-relationship model. An automated development tool, the DCDS Support System implements the DCDS database, provides an interactive form-based user interface and query capability for the database, and provides a graphics interface for generating flow diagrams.

The Software Requirements Engineering Methodology (SREM) database is structured by the Requirements Specification Language (RSL) described in part by Figure 2.14. This database provides both standard data dictionary information

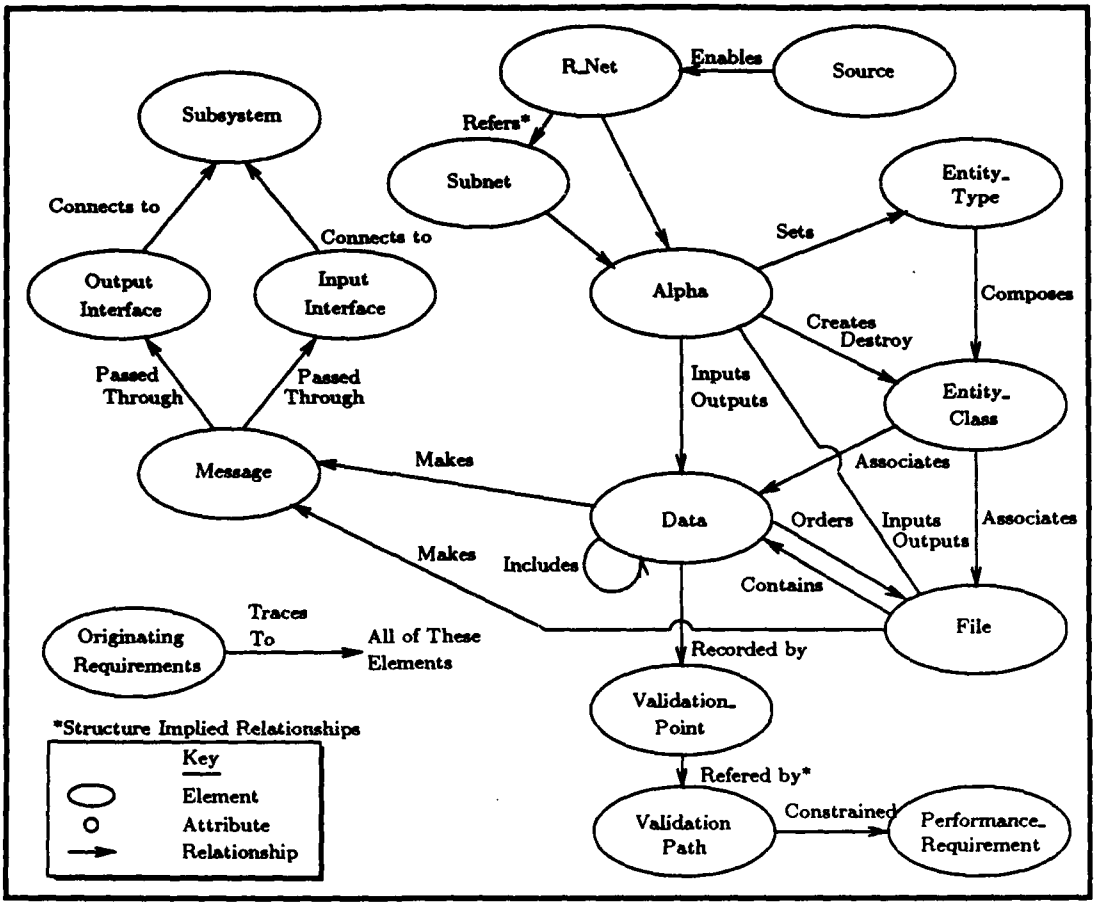


Figure 2.14. Major RSL Elements [46]

and adds the relationship terminology to generate a complete functional specification document through the DCDS Support System's powerful query capability.

A SREM software specification is centered around the requirements networks, or *R_Nets*, which identify data flow through functional processing steps called *Alpha* nodes. *R_Nets* are independent processes which are enabled upon receipt of Messages via Input_Interfaces. SREM provides a means of graphically depicting an *R_Net* as shown in Figure 2.15.

R_Nets and their *SubNets* can be developed strictly through database entries or through the graphics tool. Both graphic and textual representations can be dis-

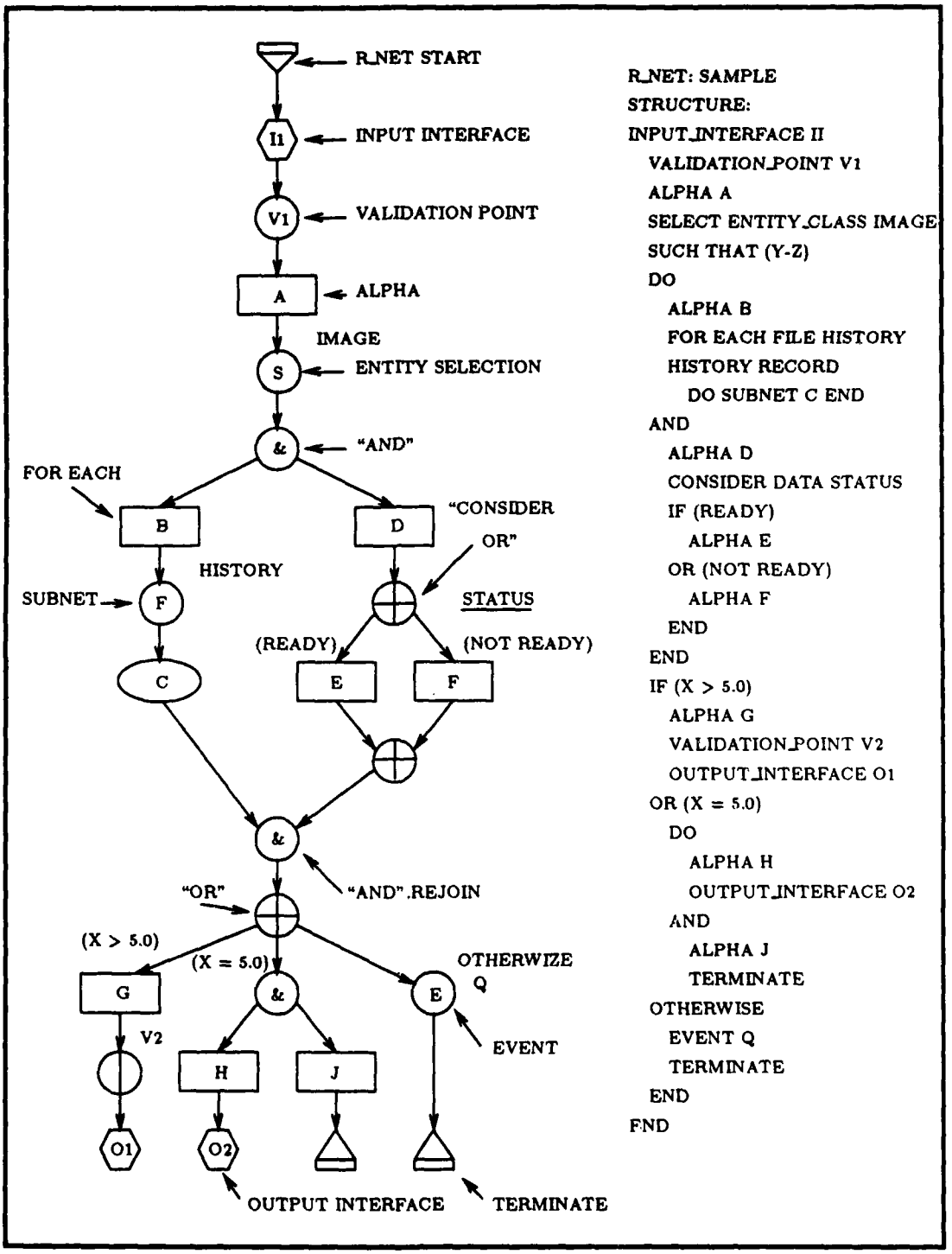


Figure 2.15. A Sample R_Net [46]

played and printed. The combination of graphic data-flow diagrams (*R_Nets*), data dictionary, and text provide a complete view of the system requirements in terms of our general requirements specification model.

SREM specifications provide elements which can be used to develop an object-oriented design. *Entity_Classes*, *Entity_Types*, *Interfaces*, *Data*, and *File* elements might be used to determine system objects and attributes. Since processing functions map to *R_Nets*, groups of *R_Nets*, *SubNets*, and *Alphas*, these may suggest candidate operations. Additionally, *Interfaces* and *Messages* are sources of visible interface descriptions.

We reiterate that there is no magic formula for mapping requirements elements to design. We asserted early on that design is a decision making process requiring judgement and choice. While automated tools, or even expert systems might aid the design process, such aid must be in the form of providing the right information, in a user-friendly manner, to help the designer make *good* design decisions. In the next section we will present the basic concepts of decision support systems (DSS) used in subsequent chapters to describe our development of a *Decision Aid for Object-Oriented Design.*

2.6 *Decision Support System Techniques*

2.6.1 Introduction. As much controversy exists as to the definition of decision support systems as to that of the object-oriented paradigm. Valusek [49] defines DSS as "a system (manual or automated) that supports the cognitive processes of judgement and choice." Ting-Peng Liang [30] adds structure to that basic concept by describing DSS as follows:

> A computer-based decision support system (DSS) is designed to improve unstructured or semi-structured decision making. It has three major components: an interactive user interface, a database management system, and a model management system.

These definitions are useful both for revealing the applicability of DSS to designing software, and for prescribing an approach to the development of a DSS for OOD.

Structured problems are those which have stable and identifiable components, easily quantifiable goals and evaluation criteria, and known constraints, assumptions, and algorithms for their solution. On the other hand, unstructured problems require intuitive inputs, require a large search space, involve uncertain parameters, and have no absolute solution.

We postulate that design has some elements of both types of problems; that it requires enough intuition and judgement in choosing between alternative solutions to be termed a "semi-structured" decision process. It is with this claim that we choose the concepts of decision support systems as a framework for developing a support environment for object-oriented design. The rest of this chapter is devoted to describing the approach used to build decision support systems. In later chapters, we describe the requirements, design, and construction of a decision aid for OOD using these principles.

2.6.2 The Design Framework. In *Building Effective Decision Support Systems*, Sprague and Carlson [44] view a DSS from three levels: the *user*, the *designer*

or analyst, and the *builder* or toolsmith. The user is the prime decision maker, the designer analyzes user requirements and specifies the high level requirements and design of the decision aid, and the builder uses computer systems hardware and software components to develop a system.

The user of a DSS is most concerned with the system's performance in supporting the decision process. Herb Simon [43] describes a model of that process characterized by the three steps of *intelligence, design*, and *choice*. Intelligence involves searching raw data for potential decisions; design requires developing and analyzing alternative courses of action, and choice is the selection of a particular action from those available. To the user, a DSS must support all three of these phases, plus the *implementation* of the final decision. But above all else, it must be easy to use [44].

Due to the unstructured or semi-structured nature of the decisions a DSS must support, highly structured software requirements methods don't work. Users are either unwilling or unable to state requirements in advance. In response to this problem, Sprague and Carlson [44] provide an analysis and design approach to eliciting DSS capabilities in terms of the following four user-oriented entities:

Representations that decision makers use to conceptualize and communicate the problem or decision situation,
Operations to analyze and manipulate those representations,
Memory Aids to assist the user in linking the representations and operations, and
Control Mechanisms to handle and use the entire system.

The builder of the DSS must decide which hardware and software components to use to construct the system. As mentioned previously, these components fall into the categories of dialogue, database, and modelbase.

Valusek [49] has combined the aspects of the three views of DSS in the three dimensional cube of Figure 2.16. This *DSS Cube* depicts the use of ROMC in translat-

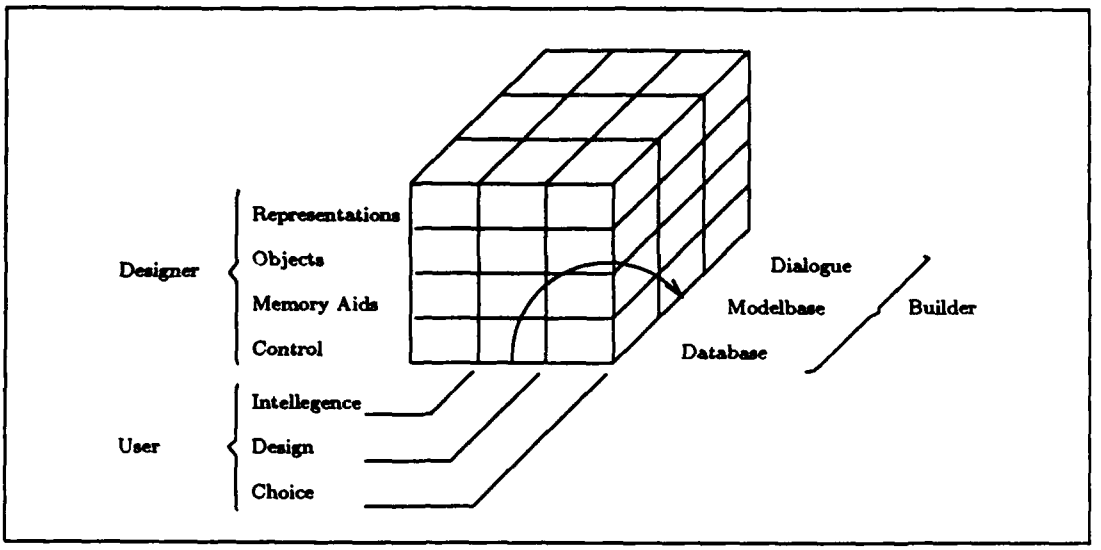


Figure 2.16. The DSS Cube. [47]

ing from the user's world into the builder's world, or in implementing user activities with builder components.

2.6.3 Adaptive Design. We mentioned that one of the key problems with DSS development is the inability to acquire complete or accurate requirements. For this reason, many in the DSS field are espousing an adaptive or iterative design approach. Rather than require the complete specification of a full-blown system, an adaptive technique begins with a kernel system, implements it, then lets it grow to meet the user's needs as determined through actual use.

This method is similar to a prototyping approach except that the prototype is intended to be used, not merely show proof of concept and thrown away. Peter Keen [25] describes it as a *middle-out* approach which relies on quickly delivering an initial system to which users respond and thus clarify their real needs.

But before even an initial system can be delivered, designers must have a way of determining basic system requirements. Keen [25] calls for beginning with a "descriptive map of user processes." Early in this chapter we introduced a method

called *concept mapping* as a means of understanding ideas, concepts and propositions. McFarren [48] has proposed using this technique during interviews with multiple users to gain a unified view of the decision processes.

Once key decision processes are identified from the concept maps, analysis is required to determine the set of DSS features which will satisfy the support requirements of those processes. Seagle and Belardo [41] propose a synthesis of the ROMC model and Structured Analysis called a *feature chart.* It serves the purpose of defining tasks and showing interfaces, as well as providing a navigational guide through the system.

After the tasks are defined, they must be modeled and then designed and implemented. To model the tasks, a series of *storyboards* can be developed which represent the functions the system may perform when fully implemented. The best presented storyboards are computer-based, with some interactive controls, communicating to the intended user the feel of what the operational system will be like. Such models are easily modified at the user's request so the designer can be reasonably sure of the validity of the requirements they represent. Thus the storyboards themselves serve to define the requirements for the system.

Given an easily adaptive dialogue component for developing the storyboards, the dialogue controls merely need to be extended to provide access to and manipulation of the required modelbase and database to produce a prototype.

A final adaptive concept has to do with how the user gives feedback to the designer regarding system problems or modifications. The solution to this problem is an on-line tool called the *hook book* [49]. The hook book is built into the dialogue component of the DSS and allows the user to immediately log problems or suggestions for modification to the system—as they come to mind through use of the tool. Hook book entries are stored in the system's database and retrieved by the designer as data points for needed changes to the system.

2.6.4 The Utilization-Shapped Evaluation Model. Riedel and Pitz [38] see evaluation as facilitating or guiding design, and consequently as an integrated process throughout the development life cycle. They address the question of what is done with evaluation results once generated by basing their *USE* model on the use of evaluation information. The focus is not on measuring the final impact of the completed system, but on who will make what decisions given feed back from the evaluation process throughout the development life cycle.

This approach seems to promise reduced cost and increased benefit. It eliminates useless evaluation criteria, thus reducing cost. It also helps designers, users, and policy-makers make decisions which may produce a better system or prevent production of a bad one—benefit. The realization that certain windows of opportunity for decision-making open and close throughout the development life cycle, and therefore gearing the evaluation plan to the decisions and decision-makers involved at those times is the unique aspect of the USE model.

The USE model is centered on the following four main concepts:

1. Select evaluation methods, measures of effectiveness, and measurement techniques based on mission requirements and DSS development/ deployment techniques.
2. Use a life cycle rather than after-the-fact approach to evaluation.
3. Consider the appropriateness of the DSS for the task it is designed for at each stage of its development (i.e., before proceeding on to further development).
4. Relate system performance to performance requirements establixhed by the system's mission.

The USE model provides the following benefits of an evaluation framework. It is

- comprehensive,
- easy to use and understand,
- able to provide a basis for considering and selecting evaluation methods and procedures,

- able to produce evaluations that provide useful information for decision making throughout the development life cycle.

In Chapter V we discuss the application of the USE model to evaluation of the decision aid and the OOD methodology.

III. An Object Oriented Design Methodology

Webster defines a methodology as "a body of *methods*, *rules*, and *postulates* employed by a discipline: a particular procedure or set of procedures" [51]. In the previous chapter we offered a number of postulates or OOD concepts; among those, an object model for supporting design. In this chapter we will reiterate briefly those postulates key to an OOD methodology, describe a method of arriving at an object-oriented design, and present several rules or heuristics applicable during the various design steps. Finally, we will review evaluation criteria for the methodology and provide a sample problem designed via the methodology.

3.1 Postulates

The basis for an object-oriented design methodology is our view of what an object is. From the previous chapter, we restate our object model definition as follows:

- An *object* is a unique entity defined by attributes which serve to identify the object and relations which associate it with other objects, attributes, and operations. Required attributes are name, behavior domain, and class. Relations include sets of operations, components, actors, and servers.
- An *attribute* identifies an object or operation.
- An *relation* represents an association of an object or operation with other system objects, operations, or relations.
- An *operation* is the description of how an object performs some behavior. Required attributes are name and algorithm. Relations include sets of actors, servers, arguments, and modified objects.
- A *class* is a complete design of an object which may be used as a template from which other objects derive their characteristic structure and function.

From this definition and the earlier discussion of OOD, we also state the following presuppositions regarding development of an object-oriented design methodology.

- Design is a decision process requiring intuition, judgement, and choice between alternatives. Design involves a set of principles and/or heuristics that guide evolution of the design, and a set of criteria upon which the final design may be judged. The objective of design is to create a representation of a system at a level of detail such that it can be built. As such, a design methodology must identify and support the decisions a designer must make as well as the creation of the representation itself.
- Object-oriented design is the process of creating a representation of a system in terms of the entities that exist in the problem space of that system. As such, it is a partial lifecycle process requiring previous analysis of the proposed system's requirements and subsequent implementation of the design in a programming language.
- The requirements specification from which the object-oriented design is developed will consist of a textual functional specification, a data dictionary, and a description of the flow of data or control through the system. The paradigm upon which such specification is based is irrelevant, as long as the specification is sufficient to fully describe the system's static and dynamic requirements.
- Since object-orientation is a qualitative assessment, and all programming languages can be said to be object-oriented to some greater or lesser degree [1], a general methodology for OOD must be language independent. Albeit the more object-oriented the language, the more straight-forward the implementation.
- An object-oriented design specification must consist of a description of each system module in terms of the object model and its dependency on the other modules in the system—at a particular level of detail. Module interface descriptions must depict which operations of an object are invoked by each operation of the object's dependent objects. In addition, the dynamic behavior of an object exhibiting a particular operation must be shown by a state diagram, flow diagram, psuedocode, or other appropriate means.
- Entities or objects represented in each system module are defined in terms of the assertions which may be made regarding them and their behaviors given certain defined stimuli. This takes the practical form of associating with each object a unique identity, the set of objects it has some relation to within the system, and the operations it requires or suffers of such objects.

[1] For example, even in assembly languages, statements consist of op-codes or operations which act on operands or objects.

3.2 Methods

We claim no special revelation as to the *right* methodology for OOD, and, in fact, relied heavily on the work of Abbott [1], Booch [9], EVB [19], Cherry [14], Lorensen [31], Seidewitz [42], and others in developing our own methodology. Our proposition is that the methodology should start with a firm foundation on previous research, but be adaptive to new ideas.

Presupposing OOD to be a decision process, we first determined the decisions required, then derived the specific steps from those decisions. We did not attempt to restructure the natural design process; rather we used the concept mapping technique described in Section 2.1.1 to derive the decision processes *from* those methods and from software engineering experts at AFIT. The resulting methodology follows the same general flow of most other design methods.

Synthesis of the various approaches to object-oriented design described in the previous chapter yielded the concept map in Figure 3.1. From the concept map, we identified the following main decision steps required to translate a requirements document into our object-oriented design specification.

1. **Analyze** the problem and requirements specification to determine a strategy for its solution.
2. **Identify** the abstract objects, operations, and attributes from the solution strategy and requirements specification.
3. **Encapsulate** the objects, operations, and attributes into modules and determine the relationships, or interfaces, between those modules. Modules should then be classified according to structure and behavior.
4. **Decompose** complex modules by repeating the process with objects or operations as separate problems, or begin detail design. Detail design proceeds as construction of modules from known components such as other objects, library modules, predefined functions or data types, or as producing an algorithmic description such as psuedocode or flow diagrams.

OOD is unique in respect to what we're looking for in our analysis of the problem, how we encapsulate data and algorithms in system modules, and in how

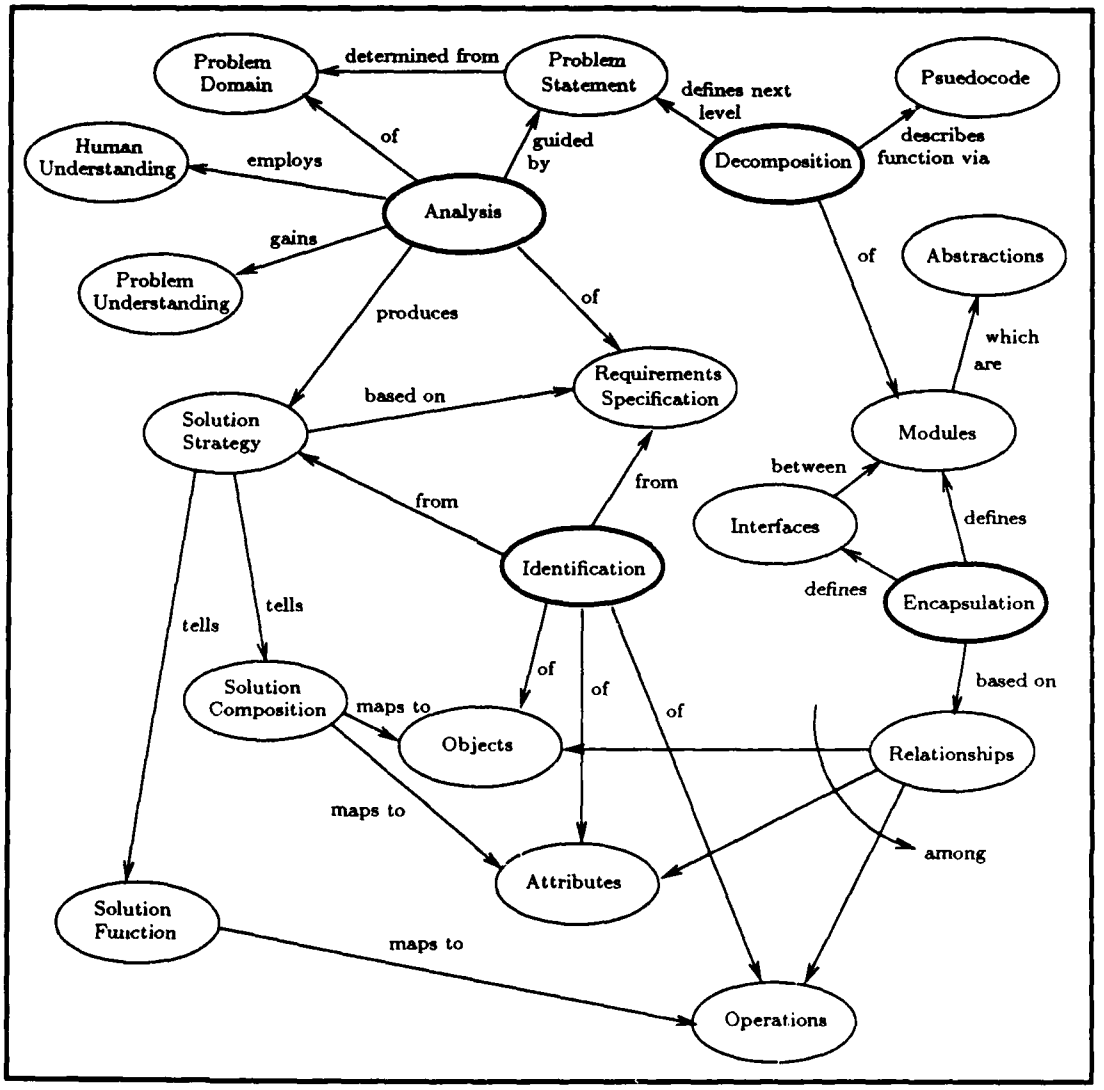


Figure 3.1. The Relationship Between Object-Oriented Design Decision Steps

we can construct system modules from known, more general data types. However, it soon becomes clear that the decisions involved are basically the same as those found in any software design methodology—regardless of the paradigm involved.

3.2.1 Analyze the Problem to Determine a Solution Strategy

3.2.1.1 Discussion. The first decision the designer must make is in limiting the scope of the problem to be solved. In this step we set the initial context or scope for the subsequent steps. We agree with Abbott [1] and others [19] that the problem must be reduced to a single sentence. A problem too complex to state in a single sentence simply requires a higher level of abstraction.

The problem statement should be determined from the problem space and stated in user-oriented terminology. One of the problems in design is the isolation of the designers from the users. Even the analysts who have developed the requirements document are normally not the users, so merely determining a design from the specification is insufficient. Interaction between designer and user is recommended for making this decision.

We feel the *concept map* may be an excellent tool for eliciting such problem-oriented information. Both the users and the high-level requirements spelled out in the specification may be used to develop concept maps. The various results may then be compared and refined to provide a clear understanding and statement of the problem. Working with the concept map of the problem, a map for a solution may be developed. We feel the concept map may be a better means of presenting the solution strategy than the single paragraph proposed by Booch [9] and others, just as the graphical structured specification [15] has been proven more effective at communicating high level abstract requirements than a verbose textual document

3.2.1.2 Summary of the Analysis Step.

1. Interview one or more users and develop concept maps of the problem.
2. Develop additional concept maps from the portions of the requirements specification which describe the system's highest level functional requirements and entities.
3. Synthesize from the concept maps a single sentence statement of the problem.
4. Develop a single concept map which depicts a strategy for solving the problem.

3.2.2 Identify the Objects, Attributes, and Operations

3.2.2.1 Discussion. Dave Bullman [27] states that finding the right objects is hard. He goes on to say that associating operations with the right objects is even harder. The implied requirement of intuition and choice here indicates this as the next decision process.

A number of "rules of thumb" or heuristics have been suggested for both the identification of objects and encapsulation of objects with their attributes and operations. Thus this step consists of the application of such heuristics to identify and define the objects, attributes, and operations which apply within the scope and level of abstraction we are dealing with. Several such heuristics are described in detail in Section 3.3. We list them here for completeness:

Object Selection Criteria lists requirements for *good* objects.
Grammatical Analysis makes selections based on nouns and verbs.
Abstraction Analysis makes selections based on data flow diagrams.
Class Abstraction makes selections based on classes of physical objects.
Concept Analysis makes selections based on concept map entities.

The primary objective of this step is *identification* along with some basic definition. We reserve associating objects and operations until the encapsulation step. The elements in this step should come initially from the solution strategy unless the heuristics used require otherwise. It is difficult to initially scope a problem such that

the list of objects, attributes, and operations is complete, accurate, and without some spurious low level objects or operations defined. Normally the analysis and identification steps will be repeated several times to arrive at a realistic scope of the problem and a complete set of objects and operations.

As identifiers of objects and operations, attributes should be associated with appropriate entities after they are identified. Listing object and operation attributes, then, serves to define those entities in greater detail. In most cases the requirements document will need to be consulted to fully describe program entities.

3.2.2.2 Summary of the Identification step.

1. Apply one or more identification heuristics to identify the set of objects in the system at the scoped level of abstraction.
2. Analyze each object and describe its attributes and structure in the solution strategy. Check the requirements document for completeness and eliminate redundancy.
3. Apply one or more identification heuristics to identify the set of operations performed within the system at the scoped level of abstraction.
4. Analyze each operation to determine its stimulus/response attributes.

The end product of this step should be a list of objects and a list of operations, with attributes describing their structure and effects.

3.2.3 Encapsulate Objects, Attributes, and Operations into Modules

3.2.3.1 Discussion. Deciding which operations should be associated with which objects is not as straight-forward as it may seem. Objects seldom behave independently of other objects. Consequently, observed behaviors may represent a complex interrelationship among objects. In the example quoted in Section 2.3.1.2, whether the operation *drill_hole* is an activity of the drill press, drill bit, or sheet metal depends on the abstraction of those objects in the problem solution. Thus guidelines, rules, or heuristics are needed to guide the encapsulation of objects and operations in such a way as to produce *good* modules.

In choosing which objects and operations to encapsulate into modules, the interrelationships between modules are revealed. We specify those relationships or interfaces by first determining the dependency between modules. A dependency exists whenever an operation of an actor or agent type object affects or requires an action by some other object. Rather than depict the dependencies only, we require diagraming the specific operations of an object required by each operation of each external object. This includes identifying the attributes or arguments an operation requires to accomplish its function, and which attributes or internal objects are affected through such an operation under the stated conditions.

The heuristics for encapsulation are described in detail in Section 3.3 and are briefly listed as follows:

Modularity Rules include rules defining quality assessment of modules such as coupling and cohesion.

Object Classification requires identifying an object's operation as one of eight general types.

Application Classification requires identifying an object's operation as one of a set of types specified as common to the program application area.

Structural Classification requires identifying a object's structure as one of four general types.

3.2.3.2 Summary of the Encapsulation Step.

1. Apply one or more encapsulation heuristics to the lists of objects and operations to determine a set of system modules.
2. Determine the interrelationships between modules and diagram the module dependencies.
3. Analyze each module dependency to determine the detailed interfaces between each dependent module's operations and the executors of those operations.
4. Refine the descriptions of the operations of each object in view of the various conditions under which it might be required of some other object.

The end product of the encapsulation step will be a set of modules defined by the object model, a module dependency diagram, and a module interface diagram.

3.2.4 Decompose the Modules or Begin Detail Design

3.2.4.1 Discussion. Decomposition deals with the question of how to construct each module. Should it be further decomposed, constructed from known components, or algorithmically defined via psuedocode or flow diagrams. This is the step in which we apply inheritance since, at this point, we have a full description of each object at a particular level of detail. To apply inheritance any earlier might result in shaping our solution to a set of preconceived notions rather than really solving the user's problem.

Inheritance is applied as we consider the object or module classifications made in the previous step. Such classifications are helpful, not only in determining module structure and behavior, but in identifying objects as instances of classes in the system, or as matching preexisting templates maintained in a class library. The decision to use inheritance is always a tradeoff between the cost of new development and the cost of modifications to existing templates.

Should inheritance fail to provide a solution to the design of a particular module, the module must be decomposed into smaller modules, or described at its lowest level as data structures and algorithms. Algorithmic description follows the traditional methods of using a structured English psuedocode or flow diagrams. Data structures which are operated on as a whole may be further described in a data dictionary.

All or part of a module may be decomposed. A module containing sets of objects and a set of operations, may have elements of those sets at their lowest level, and other elements of sufficient complexity to warrant decomposition.

Decomposition may take a variety of forms depending on the problem. For a functionally cohesive operation on a single object, conventional functional decomposition may be adequate. If aspects of the operation exhibit concurrency, a process-oriented approach may be better, with each sub-operation representing a

single concurrent operation. Should the existence of other objects become apparent, an object-oriented approach might be better. In other words, the problem should lead to an appropriate design technique, rather than squeezing the problem into an unnatural methodology.

3.2.4.2 Summary of the Decomposition Step.

1. Analyze the modules in the system for signs of common classes. If such a class hierarchy is apparent, indicate objects as instances of the class and further design the class.
2. Analyze the classification of modules in regard to existing generic structures or functions. Determine unique characteristics of such modules to determine cost effectiveness of redesign versus reuse.
3. Analyze the complexity of remaining modules and determine which module components must be further decomposed.
4. For each component which must be decomposed, determine the appropriate design method and proceed with the design. Appropriate flow diagrams, petri nets, structure charts etc. should be used to describe the design of components not accomplished in an object oriented fashion. Those components which require an object-oriented design, should be treated as new problems and designed using this methodology in an iterative fashion.
5. For each operation which need not be decomposed, describe its operation algorithmically using appropriate psuedocode or flow diagrams.
6. For each object or attribute which need not be decomposed, describe the data structure it represents.

The end product of this step will be class assignments of objects, low level operation and attribute descriptions, or non-object-oriented algorithmic designs.

3.3 Rules

3.3.1 Heuristics for Identification

3.3.1.1 Object Selection Criteria. We include the following set of software engineering heuristics which may be useful in evaluating the quality of object selection.

1. **Information Hiding.** Objects should act as black boxes to allow easy debugging and maintenance.
2. **Minimize Chained Operations.** The depth of operations chained in nested calls—operations which require operations of other objects which require operations of other objects which require..., should be minimized.
3. **Abstraction.** Objects should usually represent a single problem-domain entity. The types of abstraction, in decreasing preference, are as follows:
   - *Entity Abstractions* represent useful models of problem domain entities.
   - *Action Abstractions* represent generalized sets of operations which all perform similar functions.
   - *Virtual Machine Abstractions* group together operations used by some superior level of control, or which all use some subordinate set of operations.
   - *Coincidental Abstractions* package a set of unrelated operations or data items.
4. **Inheritance.** Identify objects which may be of the same or a known class as possibilities for code reuse. Examples are entities which may be represented by common data structures such as stacks, sets, collections etc.
5. **Overload Identifiers.** *Use the best term to identify entities* without using minor misspellings to differentiate between them.

3.3.1.2 Grammatical Analysis. This is the term we give to the method proposed by Abbott [1] et al for determining objects and operations from noun and verb phrases in a text document. The method requires the following steps:

1. Underline noun and noun phrases in the text.
2. List each noun or noun phrase and associate with each an identifier or eliminate it from the list as redundant or not applicable to the solution. Objects may be noted as a type or instance.[2]
3. Describe each object in terms of its attributes.
4. Underline verb and verb phrases in the text.
5. List each verb or verb phrase and associate with each an identifier, or eliminate it from the list as redundant or not applicable to the solution.

[2]Objects may be identified as types or classes if they are derived from are common nouns, instances if derived from proper nouns. Mass or abstract nouns denote measure or quantity and represent collections of objects or constraints on objects [37].

6. Associate the resulting operations with a single object from the solution set of objects.
7. Describe the operation of each operation in the context of the object it operates on.

3.3.1.3 Abstraction Analysis. The following method was developed by Stark and is described in [42] as a means of determining object abstractions from data flow diagrams. Since Abstraction Analysis is a complete design methodology in its own right, it may also repalce the encapsulation step.

1. Identify the transform center from the structured specification.
2. Identify the *central entity* from the transform center and the *abstract entities* that support it. These entities are identified by following the afferent and efferent flows away from the central entity and grouping related processes and states along these flows.
3. Recast the data flow diagram around the central and supporting entities.
4. Create an entity graph with a single *most senior* object which calls on a virtual machine consisting of the central entity and those other entities which directly support it.
5. Follow the afferent and efferent data flows from the transform center in the recast DFD, and identify additional abstract entities which support the previously defined entities.
6. Add the new entities to the entity graph in a new virtual machine layer.
7. Continue adding levels of entities to the entity graph and modifying the DFD until the ends of the afferent and efferent data flows on the original DFD are reached.
8. Add directions of control to the entity graph where the problem determines flow of control.
9. From the seniority relationships on the entity graph and the data stores on the recast DFD, determine entities that must be on the same virtual machine layer due to their mutual superiority to other entities or all depend on the same data store.
10. Note any cyclic graphs in the entity graph denote entities which must be on the same virtual machine layer.
11. Combine entities into objects which represent common dependencies or functions.

12. Determine alternative configurations of objects and choose the alternative that best balances requirements for loosely coupled objects and eliminates data and control bottlenecks.
13. List the objects, the processes each object implements, the states hidden by each object, and system considerations not shown.
14. Identify operations within an object which are called by another object and specify the data flows they pass.

3.3.1.4 Class Abstraction. This method refers to Lorensen's [31] approach derived more directly from OOP languages such as Smalltalk. Some concepts on object and operation selection from Lorensen's approach are as follows:

1. Data abstractions are the classes of the system.
2. Classes often correspond to physical objects within the system being modeled.
3. If not explicitly stated in the requirements document, the designer should determine abstractions from analogies drawn from the designer's experience.
4. Attributes become instance variables for each class. Specification of the data structures containing such attributes should be deferred until detail design.
5. Operations are the procedures for each class, and either access and update instance variables of the class or execute operations unique to the class.
6. Operations should only be defined as to their function. Internal design of operations will be designed by conventional methods during detail design.
7. If the class is a subclass of an existing class, thereby inheriting operations from it, determine if such operations need to be overridden by the new class.
8. Define the protocol to be used to invoke the operations.

3.3.1.5 Concept Analysis. This is the term we will use for deriving objects, attributes and operations from concept maps. This method has the following steps:

1. Generate a first cut list of objects from the entities on the concept map. This is possible since the concept map is developed by a designer with OOD in mind.
2. Identify from the list of objects which are long-lived and which are transient. Transient objects tend to be operation arguments or local variables. Long-lived objects tend to represent abstract state machines.

3. Identify which objects are subordinate, natural components of, or clearly attributes of other objects and note as such in the object description.

4. Identify the action words in the relationships between entities as candidate operations. Describe the behavior of these actions as to what objects are modified, what information is required, which objects invoke the operations, and what other operations might they naturally require of other objects.

3.3.2 Heuristics for Encapsulation

3.3.2.1 Modularity Rules

1. **Strong Cohesion.** Operations should only be coupled with those whose primary function is the manipulation of the object's private data structures. An object's set of objects and set of attributes must represent a single entity. Each operation should accomplish a single function.
2. **Loose Coupling.** Interfaces between objects should be kept simple, the number of parameters required to perform an operation minimized, and the use of global variables minimized or eliminated. Note however that control coupling is a frequent requirement of real-time systems of state machines where one object's state affects the behavior of another object.
3. **Eliminate Cycles.** The directed graph of an object diagram should seldom contain a cycle.
4. **Virtual Machine Layers.** Identify objects that support the system such as error and I/O handlers in virtual machine layers.

3.3.2.2 Object Classification. Bralick [10] lists eight general types an object's behavior might classify it as. They are listed as follows with our own explanation.[3]

1. **Static** a system constant.
2. **Passive** performs a function: server.
3. **Small** a self determining state machine, only reports its state.
4. **Weak** a state machine under control of another system entity.
5. **Demon** controls other Objects: actor.
6. **Interactive** affects other objects under another's control: agent.
7. **Sovereign** a state machine actor.
8. **Complex** a state machine agent.

3.3.2.3 Application Classification. Many applications themselves can be categorized into classes and so similar applications tend may define sets of common data structures. For example, the APEX methodology [2] developed primarily for aircraft avionics systems predefines the following data types:

[3]See also Table 2.1.

1. **Status**
2. **Storage**
3. **Sensor**
4. **Device**
5. **Counter**
6. **Pointer**

3.3.2.4 Structural Classification. In the Ada programming language, objects are represented by packages. Booch [9] states four possible types of packages as follows:

- **Abstract Data Type** An object which exports a type and a set of operations which may be performed on that type. The user of the object can define an instance of the specified type, then pass it as a parameter to the operations which manipulate the object and return a new instance of the object or some sub-object. A well defined set of example data structures is described in [8].
- **Abstract State Machine** An entity with well defined states and operations for changing from state to state.
- **Named Collections of Declarations** A logically cohesive grouping of objects and types. Similar in nature to Smalltalk pools, such common blocks sometimes can't be avoided and every effort must be made to make them easy to locate and to document their purpose and users.
- **Groups of Related Program Units** Booch illustrates this type of object with a set of mathematical library functions—which is sufficient to describe the reason for such an otherwise poorly abstracted module.

3.3.3 Heuristics for Decomposition. We listed the primary rules for decomposition in the step-by-step description in the previous section. Examples of additional heuristics which might be developed to aid the process include the following:

- Descriptions of alternate design methodologies and under what conditions they should be used.
- A specific syntax for structured-English or psuedocode.
- A description of a specific flow diagram methodology.
- A library of classes or reusable components design descriptions.

3.4 Evaluation of the Methodology

Our objectives for the stated OOD methodology were as follows:

1. The methodology must provide for recognized object-oriented concepts.
2. The methodology must be independent of the paradigm used to state the systems requirements.
3. The methodology must be independent of the programming language to be used to implement the system.
4. The methodology must be able to adapt to new discoveries regarding object-oriented design concepts and practices.
5. The methodology must be useful for producing a complete design specification.
6. The methodology must be easy to use.

That the first four criteria identified above are met by the methodology presented in this chapter is self-evident. The last two will require more proof. The next section is devoted to proof by example of the methodology's usefulness for producing the desired design specification. We offer the caveat, however, that the usefulness of any methodology can only be demonstrated over too broad a set of examples to be accomplished within the time constraints of this study. Such a quality may only be demonstrated by user acceptance over time.

Ease of use of the methodology must be demonstrated by a significant sample size of users and is closely tied to the implementation of the methodology in a support environment. In Chapter V, we present the results of the use of the methodology and support environment by a graduate level software engineering class as an indication of the usability of both methodology and tool.

3.5 A Sample Problem

The following example carries a sample problem completely through the object-oriented design methodology presented in this chapter. The requirements document is provided in the appendix.

3.5.1 Analyze the Problem

3.5.1.1 Concept Map the Problem from the User

The concept map of Figure 3.2 represents the user's view of the system. The user's view here is very data-flow oriented.

3.5.1.2 Concept Map the Problem from the Specification

Figure 3.3 represents the system at the top level as specified in the requirements document. This view is very hardware and functionally oriented. It specifies the components, and the functions the software is to perform.

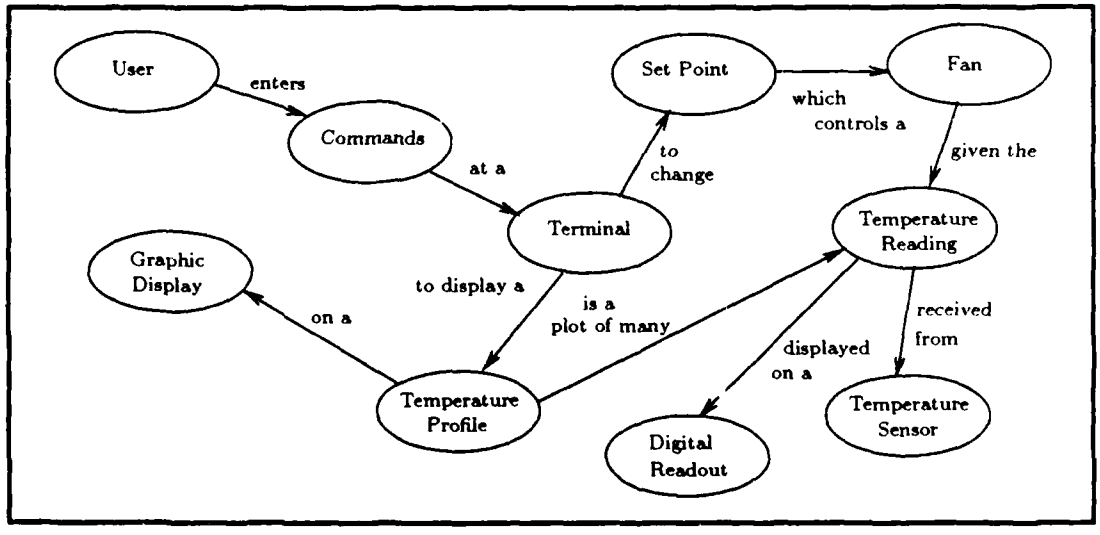


Figure 3.2. User's View of the Temperature Monitor/Controller

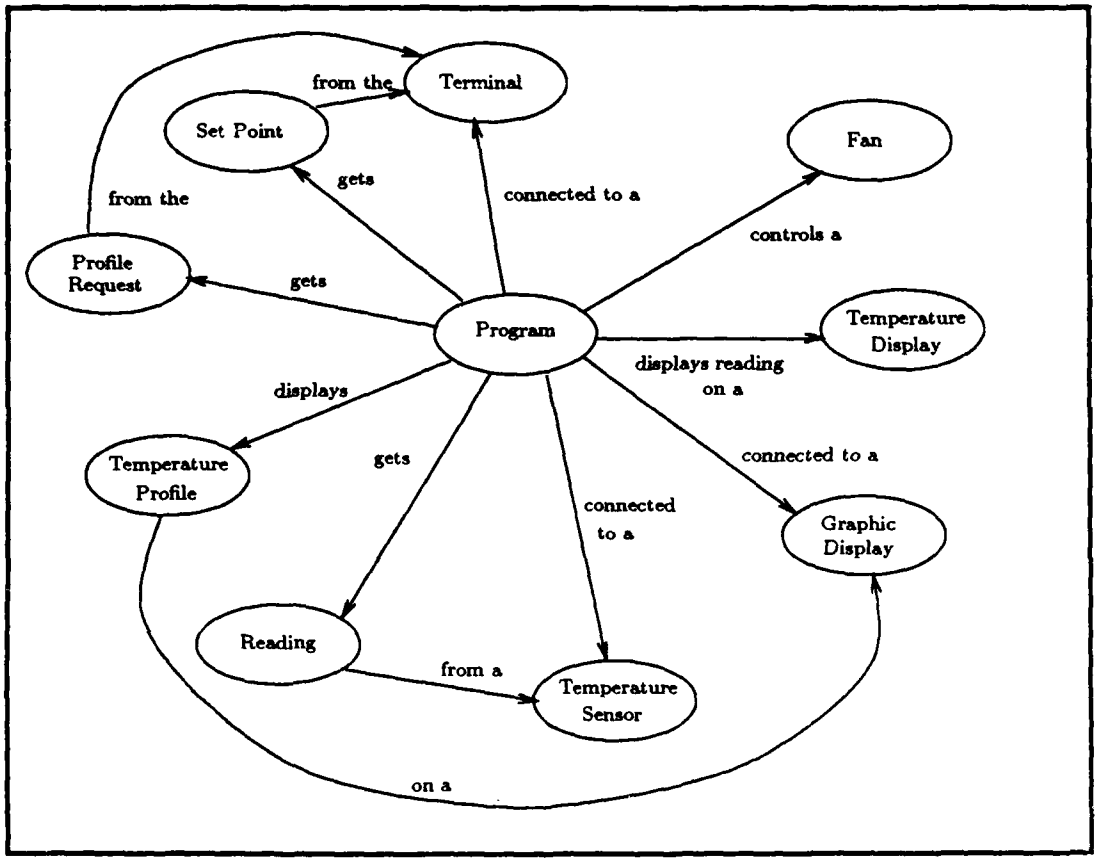


Figure 3.3. Specification of the Temperature Monitor/Controller

3.5.1.3 State the Problem

"Design a temperature monitoring and control program."

3.5.1.4 Concept Map a Solution Strategy

Figure 3.4 represents an abstract view of the system at the top level. This view is very object-oriented and describes all the objects identified in the specification concept map, as well as the interfaces between objects indicated in the user's view of the system.

3.5.2 Identify Objects and Operations

We will use *Concept Analysis* to identify and define objects and operations at the current level of abstraction.

3.5.2.1 Apply Heuristics to Identify Objects

Table 3.1 shows the objects and their analysis as determined from the solution strategy and the functional specification.

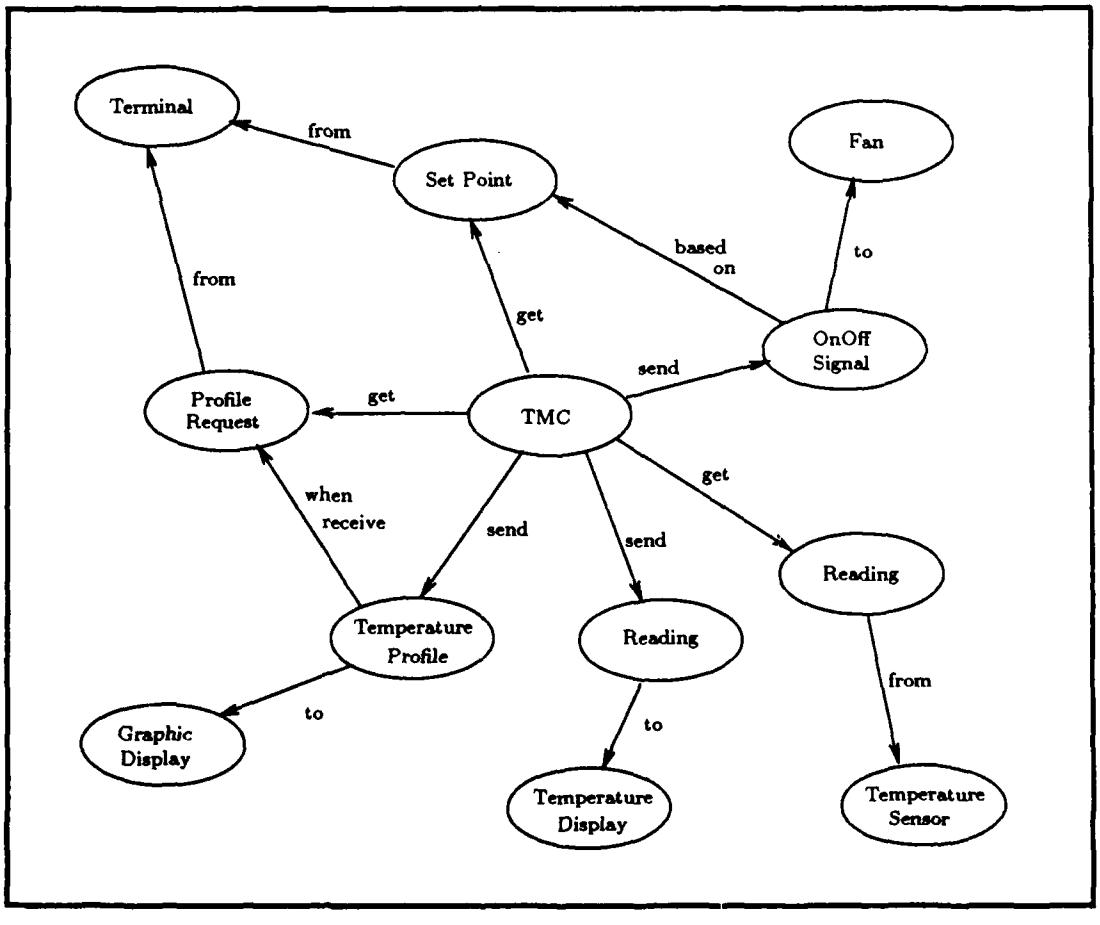


Figure 3.4. The Temperature Monitor/Controller Synthesized View

Table 3.1. Temperature Monitor/Controller Object List

Object	*Durability*	*Classification*
TMC	longlived	main program
Terminal	longlived	i/o device
Temp_Sensor	longlived	i/o device
Temp_Display	longlived	i/o device
Graphics_Display	longlived	i/o device
Fan	longlived	i/o device
Set_Point	longlived	state
Reading	longlived	state
Request	transitory	argument
OnOff_Signal	transitory	argument
Temp_Profile	transitory	argument

3.5.2.2 Describe the Objects

TMC

- Behavior: The main program.
- Component Objects: **Fan**, **Graphics_Display**, **Temp_Display**, **Terminal** and **Temp_Sensor**.
- Server Objects: I/O devices.
- Actor Objects: The System.

Terminal

- Behavior: Accepts keyboard input.
- Component Objects: **Command** (previously undefined).
- Server Objects: user inputs.

Command

- Behavior: String representing keyboard input.
- Domain: **Profile_Request** or **Set_Point** or null.

Temp_Sensor

- Behavior: Implements low level protocol to obtain **Reading** from physical device.

- Component Objects: **Reading**
- Server Objects: External I/O device.

Temp_Display

- Behavior: Implements low level protocol to send **Reading** to physical device.
- Server Objects: **Reading**

Graphics_Display

- Behavior: Implements low level protocol to display **Temp_Profile** on the physical device.
- Server Objects: **Temp_Profile**

Fan

- Behavior: Implements low level protocol to send **OnOff_Signal** to physical device.
- Server Objects: **OnOff_Signal**

Set_Point

- Behavior: Number represents maximum desireable temperature.
- Actor Objects: **TMC**
- Domain: degrees fahrenheit default: 70°

Reading

- Behavior: Number represents current temperature. **Temp_Profile**
- Domain: degrees fahrenheit.

Profile_Request

- Behavior: Represents a command to display a **Temp_Profile.**
- Actor Objects: **TMC**
- Domain: range time hhmms within last 24 hours.

OnOff_Signal

- Behavior: Represents a control signal to the **Fan.**
- Actor Objects: **Fan**
- Domain: boolean set by: **Reading** $\geq$ **Set_Point**

Temp_Profile

- Behavior: A set of **Time / Readings** pairs
- Component Objects: **Time, Reading**
- Actor Objects: **Graphics_Display**
- Domain: 8640 max elements

Time

- Behavior: Represents current system time.
- Actor Objects: **Temp_Profile**
- Domain: hhmms

3.5.2.3 Apply Heuristics to Identify the Operations

Table 3.2 shows the operations and their analysis as determined from the actions in solution strategy and the functional specification.

Table 3.2. Temperature Monitor/Controller Operation List

	Objects		
Operation	*Suffered of*	*Required of*	*Modifies*
get	TMC	Terminal	Set_Point
get	TMC	Terminal	Profile_Request
get	TMC	Temp_Sensor	Reading
send	TMC		OnOff_Signal
send	TMC		Temp_Profile

3.5.2.4 Describe the Operations

get

- Behavior: Retrieves an argument from an abstraction of an input device.
- Set of Actor Objects: (**TMC**).
- Set of Modified Objects: (**Set_Point**).
- Set of Argument Objects: (**Set_Point**).

- Set of Server Objects: (**Terminal**).

get

- Behavior: Retrieves an argument from an abstraction of an input device.
- Set of Actor Objects: (**TMC**).
- Set of Modified Objects: (**Profile_Request**).
- Set of Argument Objects: (**Profile_Request**).
- Set of Server Objects: (**Terminal**).

get

- Behavior: Retrieves an argument from an abstraction of an input device.
- Set of Actor Objects: (**TMC**).
- Set of Modified Objects: (**Reading**).
- Set of Argument Objects: (**Reading**)
- Set of Server Objects: (**Temp_Sensor**).

send

- Behavior: Commands the output device to display an argument.
- Set of Actor Objects: (**TMC**).
- Set of Modified Objects: (**Graphics_Display**).
- Set of Argument Objects: (**Temp_Profile**).
- Set of Server Objects: ($\emptyset$).

send

- Behavior: Commands the output device to display an argument.
- Set of Actor Objects: (**TMC**).
- Set of Modified Objects: (**Temp_Display**).
- Set of Argument Objects: (**Reading**).
- Set of Server Objects: ($\emptyset$).

3.5.3 Encapsulate Objects and Operations

3.5.3.1 Apply Heuristics to Determine System Modules

Discussion:

1. To the outside world, the **TMC** is a software module representing the whole system. We show it as a separate control object within the system due to represent the relationship between it as a parent of the other objects in the system.
2. The classification heuristic indicates the **Temp_Sensor** and **Terminal** are state machines with respective states of **Reading** and **Command**.
3. Encapsulating **Set_Point** with **Fan** results in a third state machine. **Fan** must, however, get **Reading** to control itself. **OnOff_Signal** becomes an internal detail.
4. Since **Temp_Display** is to be updated based only on **Reading**, it makes sense to encapsulate it within **Temp_Sensor**. Thus it becomes an internal detail.
5. All that is left is the **Graphics_Display** and the implied **Reading_Record** for the previous 24 hours. The possibly complex functionality of displaying the **Temp_Profile** and its dependence on the display device indicates the need to make **Graphics_Display** a separate module. The **Reading_Record** may logically reside in a separate module, the **Temp_Sensor** module, or the **Graphics Display** module. Since the **Graphics Display** does not naturally represent the notion of a collection of **Reading/Time** pairs of the **Reading_Record**, we choose to encapsulate the **Reading_Record** with the **Temp_Sensor**.
6. The **Temp_Sensor** now more accurately represents a **Temp_Monitor** at this level of abstraction so we will use that name.

3.5.3.2 Diagram the Module Dependencies

Figure 3.5 represents our initial module descriptions and dependencies in a block diagram.

3.5.3.3 Diagram and define Module Interface.

Figure 3.6 depicts the interfaces between modules and their operations in the system. The modules are further specified as follows:

TMC

- Set of Component Objects: (**Temp_Monitor**, **Terminal**, **Fan**, **Graphics_Display**)

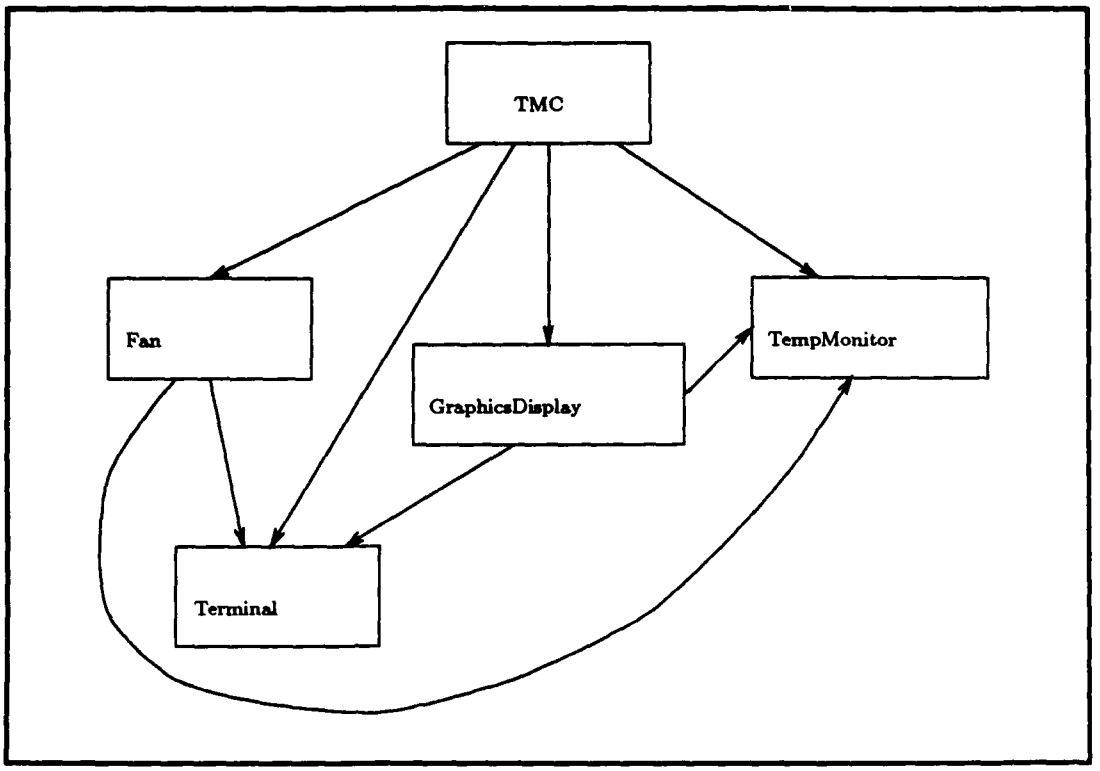


Figure 3.5. Block Diagram for the Temperature Monitor/Controller

- Set of Server Objects: (**System_IO_Drivers**)
- Set of Actor Objects: (**OS_Scheduler**)
- Set of Operations: (**run**)

Terminal

- Set of Component Objects: (**Command**)
- Set of Server Objects: (**System_IO_Driver**)
- Set of Actor Objects: (**TMC**, **Fan**, **Graphic_Display**)
- Set of Operations: (**run**, **get**)

Temp_Monitor

- Set of Component Objects: (**Reading_Record**, **Temp_Sensor**, **Temp_[4]Display**, **Reading**)
- Set of Server Objects: (**System_IO_Driver**)

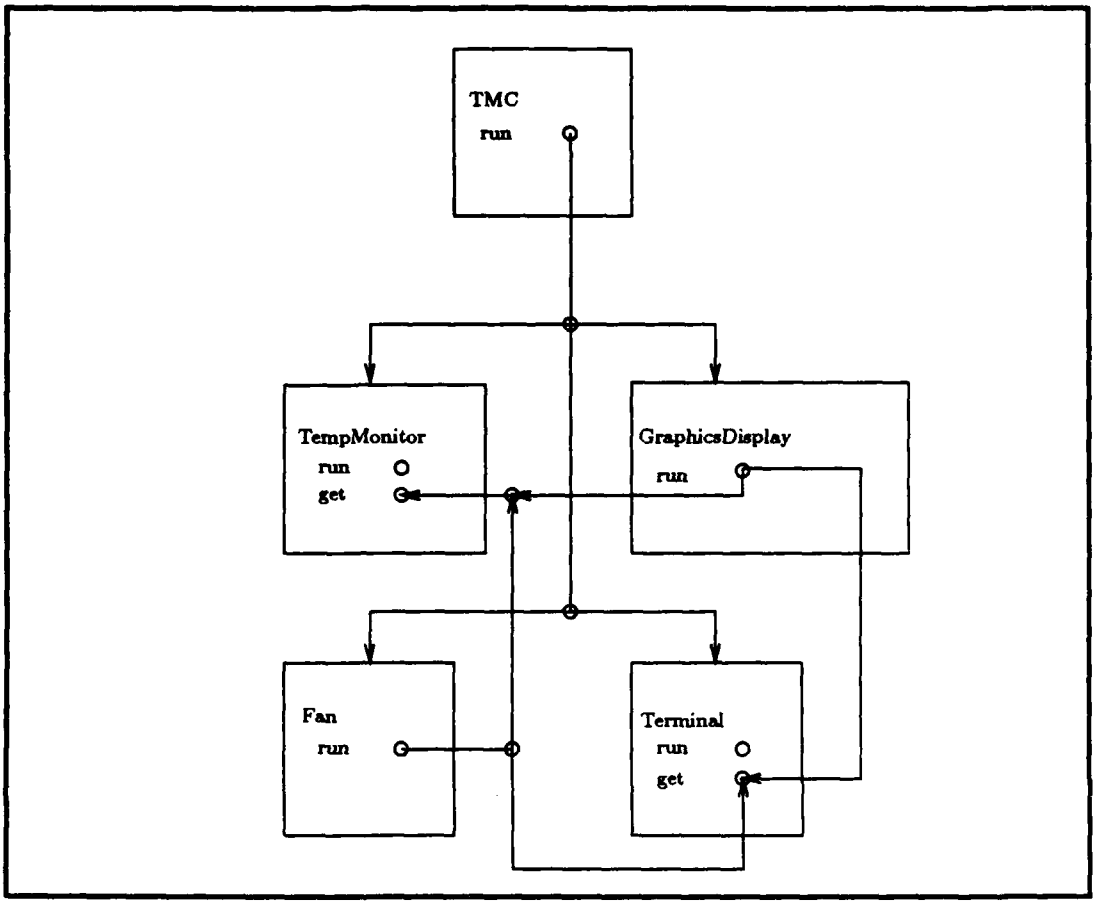


Figure 3.6. Detail Diagram for the Temperature Monitor/Controller

- Set of Actor Objects: (**TMC**, **Fan**, **Graphic_Display**)
- Set of Operations: (**run**, **get**)

Graphic_Display

- Set of Component Objects: ($\emptyset$)
- Set of Server Objects: (**System_IO_Driver**)
- Set of Actor Objects: (**TMC**)
- Set of Operations: (**run**)

Fan

- Set of Component Objects: (**Set_Point**)

- Set of Server Objects: (**Temp_Monitor**, **Terminal**)
- Set of Operations: (**run**)

3.5.3.4 Refine Module Behavioral Descriptions

TMC

1. Operation **run**
   - Set of Input Arguments: ($\emptyset$).
   - Set of Output Arguments: ($\emptyset$).
   - Behavior:

     The **TMC** causes each of its four component state machines to begin running upon execution of the system. Halting the system will result in termination of each of the component objects.

Terminal

1. Operation **run**
   - Set of Input Arguments: ($\emptyset$).
   - Set of Output Arguments: ($\emptyset$).
   - Behavior:

     The **Terminal** retains an input character buffer called **Command**. When **Command** is changed via keyboard input, it is examined to determine if it represents either a **Set_Point** or a **Profile_Request** protocol. If it does, the **Command** is maintained and a flag is set stating which type of command it is. Otherwise, the buffer is flushed and the flag is set to null.
2. Operation **get**
   - Set of Input Arguments: ($\emptyset$).
   - Set of Output Arguments: (**Profile_Request**).
   - Behavior:

     If the receiver object represents a **Profile_Request**, the **Command** is returned. Otherwise a null string is returned.
3. Operation **get**
   - Set of Input Arguments: ($\emptyset$).
   - Set of Output Arguments: (**Set_Point**).

- Behavior:

  If the receiver object represents a **Set_Point**, the **Command** is returned. Otherwise a null string is returned.

Temp_Monitor

1. Operation **run**
   - Set of Input Arguments: ($\emptyset$).
   - Set of Output Arguments: ($\emptyset$).
   - Behavior:

     The **Temp_Monitor** periodically gets a **Reading** from the physical input device and sends it to the physical output device. The **Reading** is saved in a **Reading_Record** data structure.
2. Operation **get**
   - Set of Input Arguments: (**Profile_Request**)
   - Set of Output Arguments:(**Reading_Record**)
   - Behavior:

     The set subset of **Reading/Time** pairs which fall within the **Time** range specified in the **Profile_Request** are returned.
3. Operation **get**
   - Set of Input Arguments: ($\emptyset$)
   - Set of Output Arguments:(**Reading**)
   - Behavior:

     The most current **Reading** from the **Reading_Record** is returned.

Graphic_Display

1. Operation **run**
   - Set of Input Arguments: ($\emptyset$).
   - Set of Output Arguments: ($\emptyset$).
   - Behavior:

     The **Graphic_ Display gets a Profile_Request** from the **Terminal.** When it receives one that is not null, it **gets** the appropriate **Reading_Record** set from the **Temp_ Monitor** and formats the display protocol for the output device.

Fan

1. Operation **run**
   - Set of Input Arguments: ($\emptyset$).
   - Set of Output Arguments: ($\emptyset$).
   - Behavior:

     The **Fan** periodically **gets** a **Reading** from the **Temp_Monitor**. The **Reading** is less than its internally maintained **Set_Point**, it sends a signal to the physical device to turn it on, otherwise it sends a signal to turn it off. The **Fan** also periodically **gets** a **Set_Point** from the **Terminal**. If it is not null, it updates its internal state with the new value.

3.5.4 Decompose the Modules or Begin Detail Design

3.5.4.1 Analyze the Modules for Common Classes

The state machines representing I/O devices may represent instances of a class like I/O_Drivers. However, the functionality of each seems sufficiently different to eliminate any implementation benefit.

3.5.4.2 Analyze the Modules for Existing Generic Structures

Most languages should provide interface routines, such as Text_IO or predefined pragmas in Ada, for implementing the external device I/O.

3.5.4.3 Analyze the Module Complexity

The **TMC**, **Fan**, and **Terminal** are obviously at a sufficiently low level of detail to describe with psuedocode or flow diagrams.

The **Graphic_Display** and **Temp_Monitor** may require further decomposition.

3.5.4.4 Determine the Appropriate Method for Decomposition

The **Graphic_Display**, might require decomposition if the graphics commands are not of sufficient power to easily plot the graph without having to calculate the entire bit_map. Since the functionality of the display operation appears to be primarily calculations, functional decomposition would probably be an adequate means for carrying out further design.

The **Temp_Monitor** is clearly more complex than the other objects, and could be decomposed in an object oriented fashion. However, the primary objects represent the external temperature sensor and display, and the **Reading_Record** data structure. The external devices need not be represented by separate modules since the operations and arguments involved would already presumably be defined by system calls or generic routines. That leaves only the data structure which should be detail designed by implementing an appropriate data structure such as an array.

3.5.4.5 Describe Operations of Lowest Level Modules

Since each of the specified operations may easily be described with conventional methods, we refer the reader to the appendix for the detail design. But for completeness of the three view object model representation, a petri net graph of the Temperature Monitor/Controller is presented in Figure 3.7.

3.5.4.6 Describe Data Structures of Lowest Level Modules

Table 3.3 shows the low level objects and their structural descriptions.

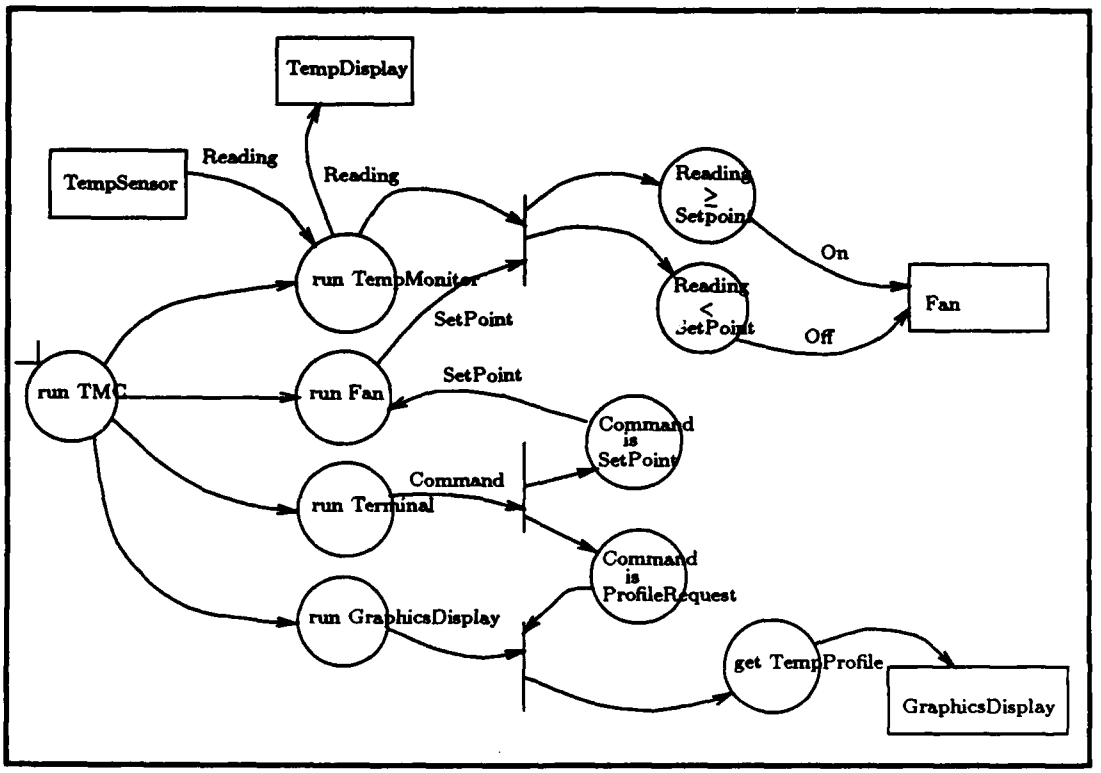


Figure 3.7. Petri Net Diagram for the Temp_Monitor Module of the TMC

Table 3.3. Temperature Monitor/Controller Data_Structures

Object	*Data Structure*	*Domain*
Command	string	character
Set_Point	numeric	degrees F. ±999.99
Reading	numeric	degrees F. ±999.999
Request	range	integer hhmms..hhmms
Temp_Profile	array(8640)[a]	Reading

[a]Since times are specified at 10 second intervals, and the size at 24 hours, the time might be deduced from the index to the array, so a single dimensional, 8640 element array would be sufficient.

3.5.5 Conclusion

The Temperature Monitor/Controller problem is a typical, although simple example of a real time system involving concurrent processes. Using the proposed methodology, we easily identified the required components, described their structures and relationships in terms of the object model, and presented the information graphically. Detail design and implementation of the problem can proceed in a straight-forward fashion via psuedo code or a specific language oriented PDL.

IV. Requirements and Design of a Decision Aid

4.1 Introduction and General Requirements

In this chapter we discusses the steps taken for determining requirements and a top level design for a decision aid to implement the previously described OOD methodology. In the field of decision support, requirements determination requires four steps: understanding the problem, selecting a kernel system to implement, developing a representation or model of the system in the form of storyboards, and describing the database and modelbase requirements to support the system. The storyboards and associated feature chart then serve as a top level design of the dialogue, database, and modelbase components of the decision aid.

Before getting to the specific requirements to support OOD, we give the general requirements for the decision aid as follows:

1. The DSS dialogue shall be a mouse driven, windowed environment.
2. The dialogue shall be easily modifiable.
3. The dialogue shall present standard capabilities in an orthagonal manner. For example, the help function shall always be accessible in the same manner, and in the same place in each of the main displays.
4. The dialogue shall support a hook book entry capability.
    a. The hook book shall require a date, circumstances, and idea of the user, and automatically maintain a unique identifier for each entry.
    b. The hook book shall be retrievable for display and hard copy.
    c. Hook book entries may be deleted or edited.
5. The dialogue shall provide an interactive help function.
6. The dialogue shall support access and control over the modelbase and database components.
7. The dialogue shall provide a unique entry point providing for initialization of the database and modelbase and introductory help capability.
8. The dialogue shall provide an exit capability from any point and shall prompt the user to decide whether changes will be saved.

9. The DSS shall allow specific selection and activation of conceptual or mathematical models from the dialogue.
10. The DSS shall provide the capability to interactively modify or add models to the modelbase from within the DSS.
11. The DSS shall allow retrieval and update to a database or multiple databases from within the DSS.
12. The DSS shall optionally allow protecting sensitive databases from modification by the user of the DSS.

4.2 Understanding the Decision Making Process

Throughout this document we have been using concept maps as a means of conveying understanding. We said they can also be used as a means of *gaining* understanding of a decision process (Figure 2.1). In Chapter II, we used concept maps to gain and described our understanding of the object-oriented paradigm, OOP and OOD, and presented general models for both requirements and design specifications. Then in Chapter III we presented a methodology for OOD based on those concept maps. Figure 3.1 shows the decision steps involved in OOD and acts as an overview for the methodology.

4.3 Selecting the Kernel.

The concept map of Figure 3.1 reveals the central decision processes of *analysis*, *identification*, *encapsulation*, and *decomposition*. Using these four main steps and their descriptions from Chapter III, we developed the feature chart and storyboards we described in this chapter.

The feature chart, Figure 4.1, depicts the support and interaction required by the four steps in our OOD methodology. Besides the four main decision steps, an entry/exit storyboard has been added to provide control over initialization and loading and unloading the design database. Features required to support methodology sub-steps have also been added to complete the systems general requirements.

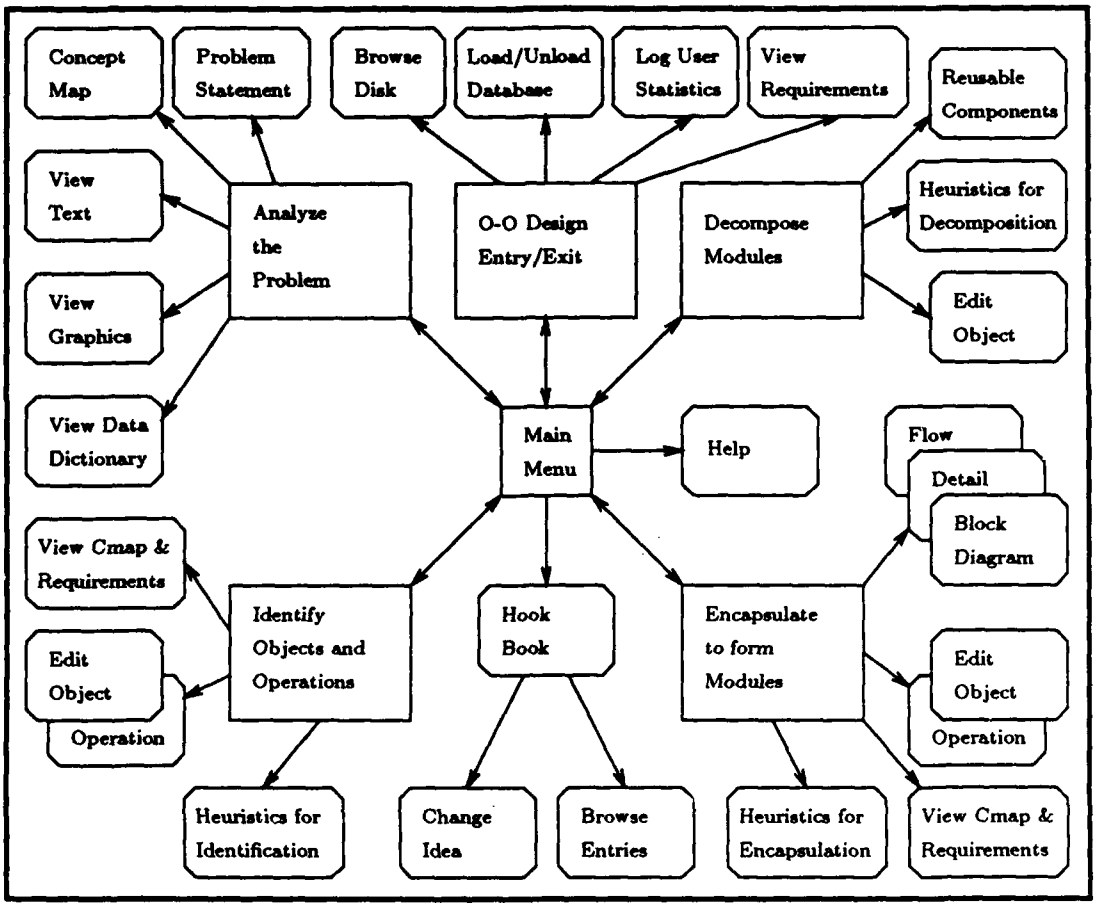


Figure 4.1. Feature Chart for the OOD Decision Aid

On the feature chart, large rectangular boxes represent main screen windows or storyboards which fill an entire screen. The small rectangular boxes represent menus used from within the storyboards to select specific actions in support of the decision step. Boxes with clipped corners represent a function of the storyboard in support of one of the substeps in the methodology.

The five storyboards are linked together through the *main menu* which will be available from each storyboard for switching to any other storyboard. The main menu will also provide for exiting the system and allow access to a context sensitive help function and the hook book. Several functions overlap. For instance, the object

and operation definitions created in the Identification storyboard are used again in both subsequent storyboards.

The feature chart presents an overview of the features required by the kernel system. Consequently, only the high level functions are shown. The storyboards and their descriptions reveal detailed requirements.

4.4 Representing the Kernel

The detail requirements for the decision aid are represented in the storyboards of this section. First we present general requirements for each storyboard—based on the methodology of Chapter III and the feature chart. Then, the storyboards themselves, along with their descriptions, are presented in the final figures of the section. In addition to the storyboards representing the five main screen displays, we also show a storyboard for the hook book.

Each storyboard contains at least three sub-windows or panes: a *features* pane, an *objects* pane, and a *text* pane. Selecting an element in the features pane will cause a list of files or objects to appear in the objects pane. Selecting an element in the objects pane will cause initialization of the text pane, or bring up a sub-window—either one of which the user will use to carry out some sub-step in the methodology.

4.4.1 Requirements for the OOD Entry/Exit Storyboard. The general requirements for the Entry/Exit storyboard are as follows:

1. The Entry/Exit display shall support the following activities:
   a. Log on to the OOD Decision Aid.
   b. Load the design database.
   c. Transfer to one of the four main OOD storyboards.
   d. Save the design database upon exiting the system.
2. The Entry/Exit display shall provide the capability to browse the disk files and change the default directory.
3. The Entry/Exit display shall prompt the user for an id and automatically log user time on the system.

4. The Entry/Exit display shall provide the capability to selectively list requirements database files.
5. The Entry/Exit display shall provide a help file giving adequate instructions for the first-time user to effectively use the system.
6. Figure 4.2 provides the Entry/Exit storyboard and its detailed description.

4.4.2 Requirements for the Analysis Storyboard. The general requirements for the Analysis Storyboard are as follows:

1. The Analysis display shall support the following activities:
   a. Concept map the problem from the user.
   b. Concept map the problem from the specification.
   c. State the problem.
   d. Concept map and state a solution strategy.
2. The Analysis display shall provide the capability to access an on-line requirements specification.
   a. Access to the specification will be in three forms: text, data dictionary, and graphical flow diagram.
   b. Access to the specification shall not corrupt the current state of currently entered data.
3. The Analysis display shall provide the capability to generate, edit, store, and retrieve concept maps.
4. The Analysis display shall provide the capability to enter, edit, store, and retrieve a textual problem statement and solution strategy.
5. Figure 4.3 provides the Analysis storyboard and its detailed description.

4.4.3 Requirements for the Identification Storyboard. The general requirements for the Identification storyboard are as follows:

1. The Identification display shall support the following activities:
   a. Apply heuristics to identify objects and operations.
   b. Analyze the solution strategy and requirements specification.
   c. Describe object and operation attributes.
2. The Identification display shall support the same specification access capabilities as the Analysis display.

3. The Identification display shall provide access to concept maps.
4. The Identification display shall provide access to identification heuristics.
5. The Identification display shall provide the capability to enter objects and operations into a database and edit their attributes.
6. Figure 4.4 provides the Identification storyboard and its detailed description.

4.4.4 Requirements for the Encapsulation Storyboard. The general requirements for the Encapsulation storyboard are as follows:

1. The Encapsulation display shall support the following activities:
   a. Apply heuristics to determine system modules.
   b. Diagram module dependencies.
   c. Diagram module interfaces.
   d. Refine object and operational descriptions.
2. The Encapsulation display shall provide access to encapsulation heuristics.
3. The Encapsulation display shall provide a graphic editor to generate, store, retrieve, and edit block and detail diagrams.
4. The Encapsulation display shall provide the capability to edit object and operation attributes.
5. Figure 4.5 provides the Encapsulation storyboard and its detailed description.

4.4.5 Requirements for the Decomposition Storyboard. The general requirements for the Decomposition storyboard are as follows:

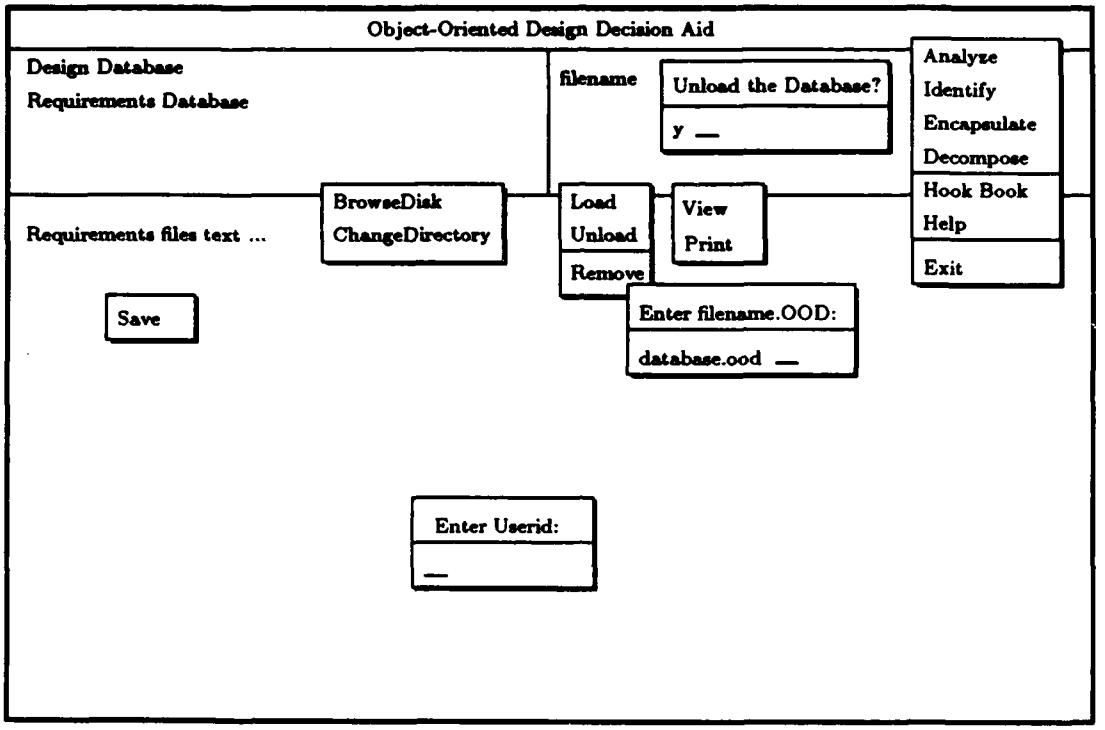
1. The Decomposition display shall support the following activities:
   a. Analysis of modules for level of detail.
   b. Analyze modules for common class or inheritance.
   c. Enter psuedocode description of low level operations.
   d. Enter data structure descriptions in the data base.
2. The Decomposition display shall provide access to graphical and database object and operational descriptions.
3. The Decomposition display shall provide access to decomposition heuristics.
4. The Decomposition display shall provide edit capability of object database.
5. The Decomposition display shall provide the capability tc identify a module to be decomposed and delineate the new level of abstraction in the database.
6. Figure 4.6 provides the Decomposition storyboard and its detailed description.

4.4.6 Requirements for the Hook Book. The general requirements for the Hook Book display are as follows:

1. The Hook Book display shall provide the capability to log ideas for changes to the decision aid.
2. The Hook Book display shall provide the capability to retrieve, edit, and print Hook Book entries.
3. The Hook Book will automatically record the user, date, time, and storyboard from which the Hook Book was called.
4. Figure 4.7 provides the Hook Book storyboard and its detailed description.

4.5 Detailed Requirements: The Storyboards

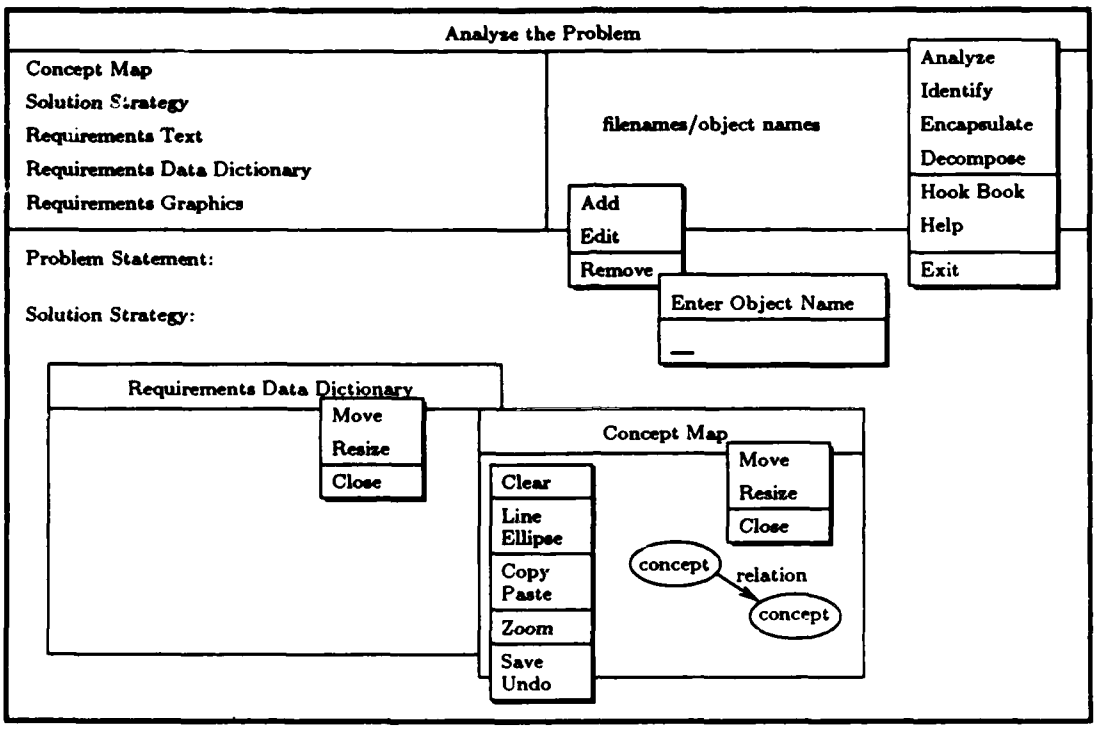
The subsequent figures present detailed requirements for the decision aid in the form of storyboards and associated descriptive text.



The Entry/Exit display is described as follows:

1. The user will initailly be prompted for a userid. Login/logout times will be automatically recorded.
2. The main menu will allow activating other storyboards, the Hook Book, context sensitive help, or exiting the system.
3. Selecting entries from the features pane produces the following results:
   a. Selecting the DesignDatabase causes database files to be listed in the objects pane.
   b. Selecting the Requirements Database causes all requirements files to be listed in the objects pane.
4. The features pane popup will allow the following:
   a. Activation of a disk browser facility.
   b. Prompting the user for a new default directory for the database, requrements, or help files.
5. The objects pane popup will allow loading and unloading the design database; or printing/viewing requirements files.
6. The text pane will provide the following:
   a. Initial instructions on startup. Editing and saving instructions.
   b. Viewing requirements files.

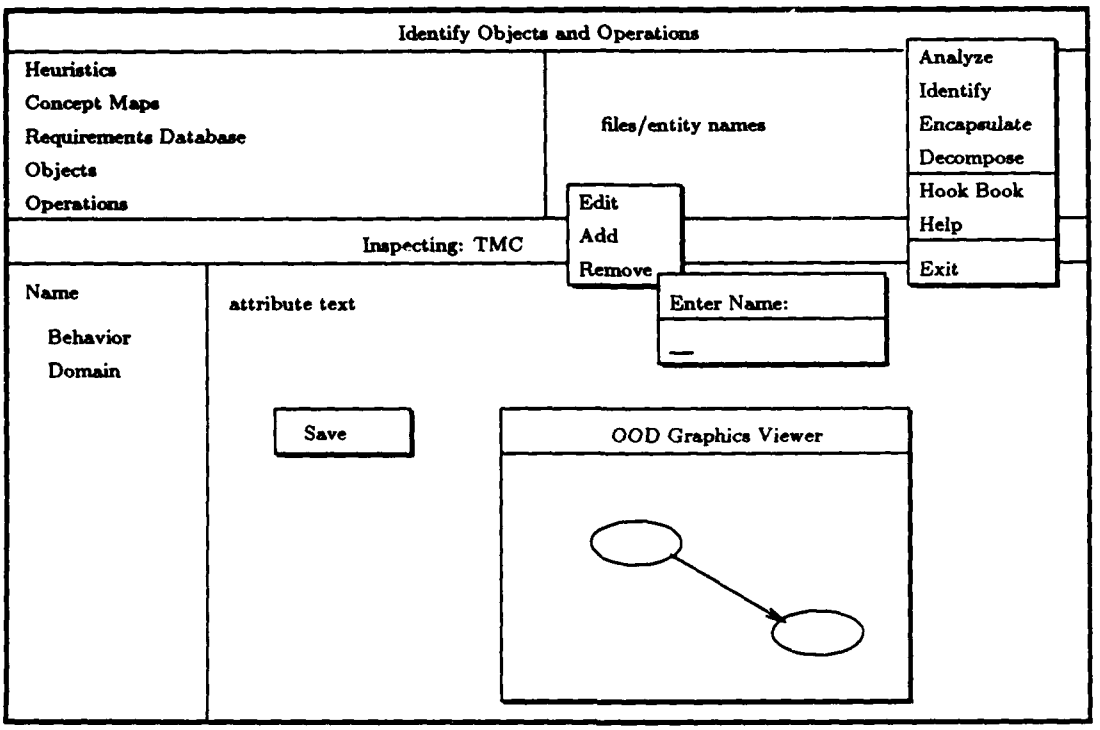
Figure 4.2. Storyboard: Entry/Exit for the OOD Decision Aid



The Analysis display is described as follows:

1. The main menu may be activated. Exit will return user to Entry/Exit.
2. The user will be prompted for the name of the object to be designed.
3. Selecting entries in the features pane results in the following actions:
   a. Selecting the ConceptMap entry causes concept map object names to be listed in the objects pane.
   b. Selecting Solution Strategy causes object names to be listed in the objects pane.
   c. Selecting RequirementsText, DataDictionary, or Graphics entries causes the corresponding files to be listed in a sub-window.
4. Selecting entries in the objects pane results in the following actions:
   a. Selecting a concept map will allow editing, saving or removing concept maps from objects.
   b. Edit will open a graphic drawing window for creating, editing, and saving concept maps.
   c. Selecting Edit will bring up a sub-window for editing the concept map.
   d. Selecting an object for a Solution Strategy will display the text to the text pane or format the text pane for creation.
   e. Selecting a requirements file will activate a sub-window for viewing requirements data.
5. The concept map sub-window will provide the capability to generate and edit graphics representations of concept maps.

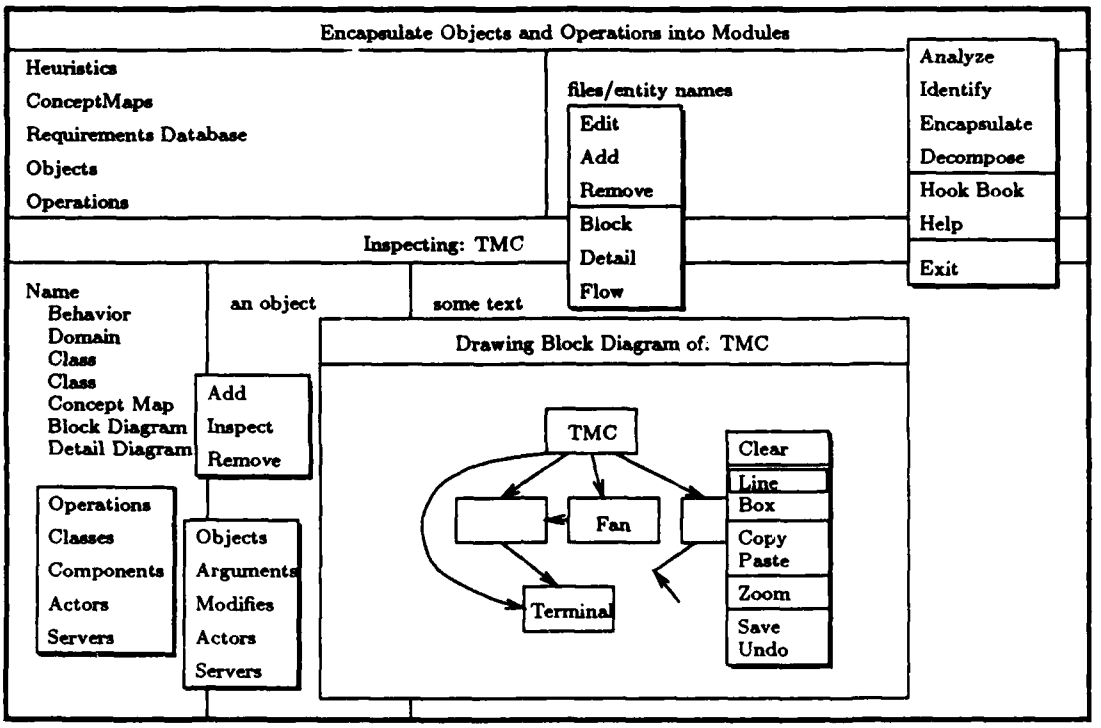
Figure 4.3. Storyboard: Analyze the Problem



The Identification display is described as follows:

1. The main menu may be activated. Exit will return user to Entry/Exit.
2. Selecting entries from the features pane produces the following results:
   a. Selecting Heuristics causes a Help window to open for viewing/editing heuristics.
   b. Selecting Requirements, or ConceptMap causes file or object names to be listed in the objects pane.
   c. Selecting Object or Operation causes database entries to be listed in the objects pane.
3. Selecting an entry in the objects pane produces the following results:
   a. Selecting a concept map, or a requirements source file, will activate the appropriate sub-window for viewing only.
   b. Selecting an object or operation name will activate a popup for Adding, Editing, or Removing entities from the database. operation.
   c. Selecting Add will open a database browser in the text window.
4. The Database Editor will provide the following capabilities:
   a. An attribute pane will provide the ability to add, inspect or remove attributes.
   b. A text pane will allow editing an entry's attributes.

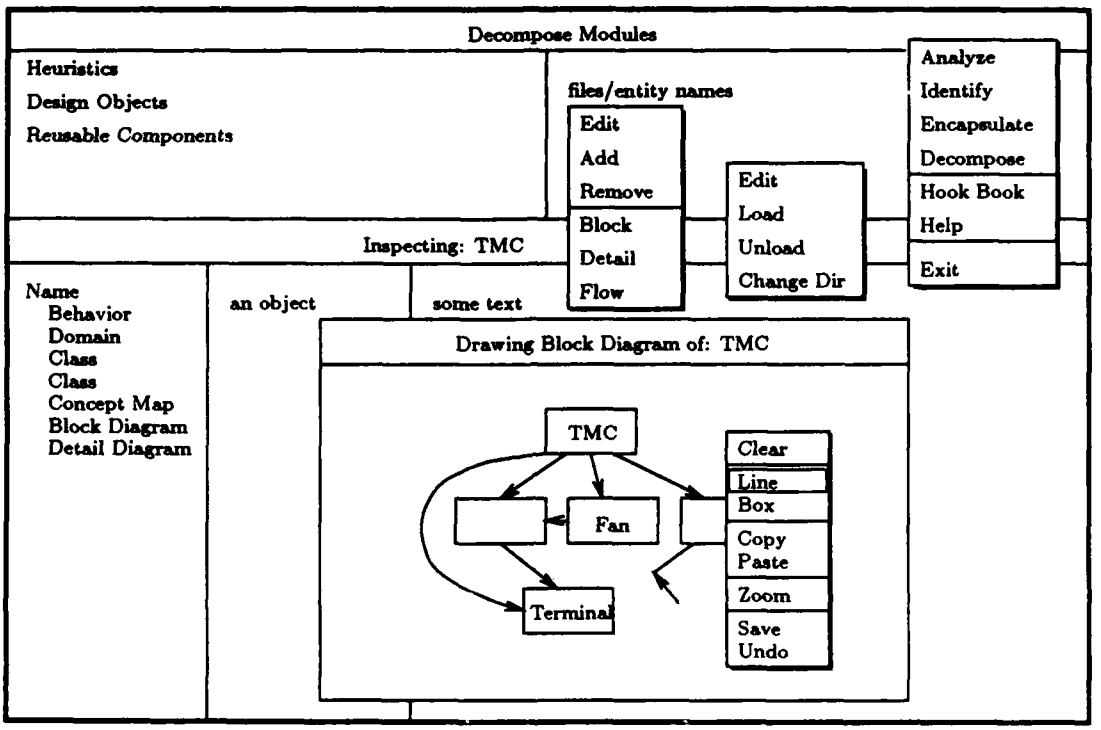
Figure 4.4. Storyboard: Identify the Objects and Operations



The Encapsulate display is described as follows:

1. The *main menu* may be activated. Exit will return user to Entry/Exit.
2. Selecting entries from the features pane results in the appropriate file or object names being listed in the features pane.
3. Selecting entries from the objects pane results in activation of the appropriate sub-window—except for objects and operations.
4. Selecting an *object or operation* from the objects pane opens a pop-up for selecting Editing the object/operation or creating Block, Detail, or Flow diagrams.
5. Selecting Edit opens a Database Browser with an additional list pane for forming relations.
6. The Database Browser provides the following additional capabilities:
   a. A context sensitive pop-up menu will list the possible relations for either an object or operation.
      Object: Operations, Components, Actors, Servers, Classes.
      Operation: Objects, Arguments, Modifies, Actors, Servers.
   b. Selecting a relation causes a second pop-up to appear for selecting Add, Remove, or Inspect.
   c. Selecting Add lists all appropriate objects or operations from wich to select in the list pane.
   d. Selecting inspect lists all defined objects or operations in the relation for the selected object. Selecting one opens an Inspector window on the object.
7. Selecting Block, Detail, or Flow results in activation of a graphics sub-window similar to the concept map sub-window.
8. Graphics sub-windows will provide for creation of rectangles or circles or other shapes as appropriate to the type of graphic being developed.

Figure 4.5. Storyboard: Encapsulate the Objects with their Operations



The Decomposition display is described as follows:

1. The main menu may be activated. Exit will return user to Entry/Exit.
2. Selecting entries from the features pane produces the following results

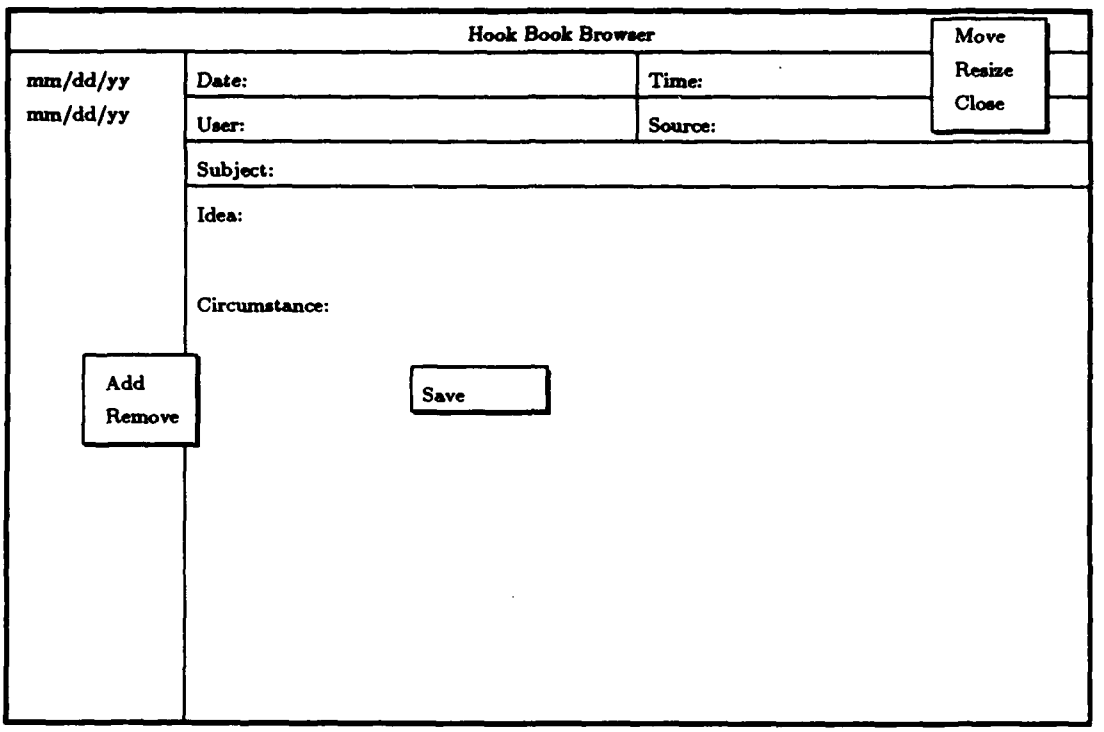
   a. Selecting Heuristics opens a help window for decomposition heurictics.

   b. Selecting DesignObjects lists objects in the files pane in a component hierarchy. Selecting objects opens a Database Browser as with the Encapsulation storyboard.

   c. Selecting ReusableComponents lists reusable components database files in the objects pane.

3. The objects pane provides a pop-up for load/unloading reusable components databases, and editing objects and their associated graphics.

Figure 4.6. Storyboard: Decompose the Modules



The Hook Book Browser display is described as follows:

1. A popup menu will provide the ability to move, resize, or close the browser.
2. A list pane will list all hook book entries by date and time.
3. Selecting an entry will cause the corresponding mini-panes in the rest of the window to be updated from the hook book entry.
4. Selecting Enter will cause the Date, Time, Userid, and Storyboard called from to automatically be entered in the labeled min-panes.
5. The user will be immediately prompted for a subject.
6. The text pane will provide for entering and saving the idea and circumstances.

Figure 4.7. Storyboard: The Hook Book Browser

4.6 Supporting the Kernel

4.6.1 The Database Requirements

The database requirements for the OOD decision aid may be divided into three categories. First will be the existing database representing the requirements specification used as a source document for design.

> • The *requirements database* to be used in the prototype shall provide a text functional specification, a data dictionary, and a graphical representation of the data flow requirements.

The second category consists of those databases supporting existing design tools, other tools such as text editors, word processors, and software configuration management libraries available in the environment.

> • The *tools database* shall provide for storage and retrieval of both text and graphical images in support of the methodology implemented in the dialogue. Text files include the help files, heuristic files, and hook book entries. Graphic images of the block and detail diagrams must be saved and indexed for retrieval.

The third category is that which supports the object model itself. The underlying object model to be used in this methodology was described in Section 2.8. An entity-relationship (E-R) diagram [29] for this model is presented in Figure 4.8. Figure 4.9 additionally gives the set of relation skeletons derived from the E-R diagram.

> • The *object model database* shall provide the capability to store and access descriptive information required by the object model.

4.6.2 The Modelbase Requirements

"Models are active relations and associations that govern decisions and actions in an organization" [28]. For our purposes, the object model of Chapter II and the heuristics and methodologies listed in Chapter III comprise the "relations and associations" which govern the design decisions in the OOD process.

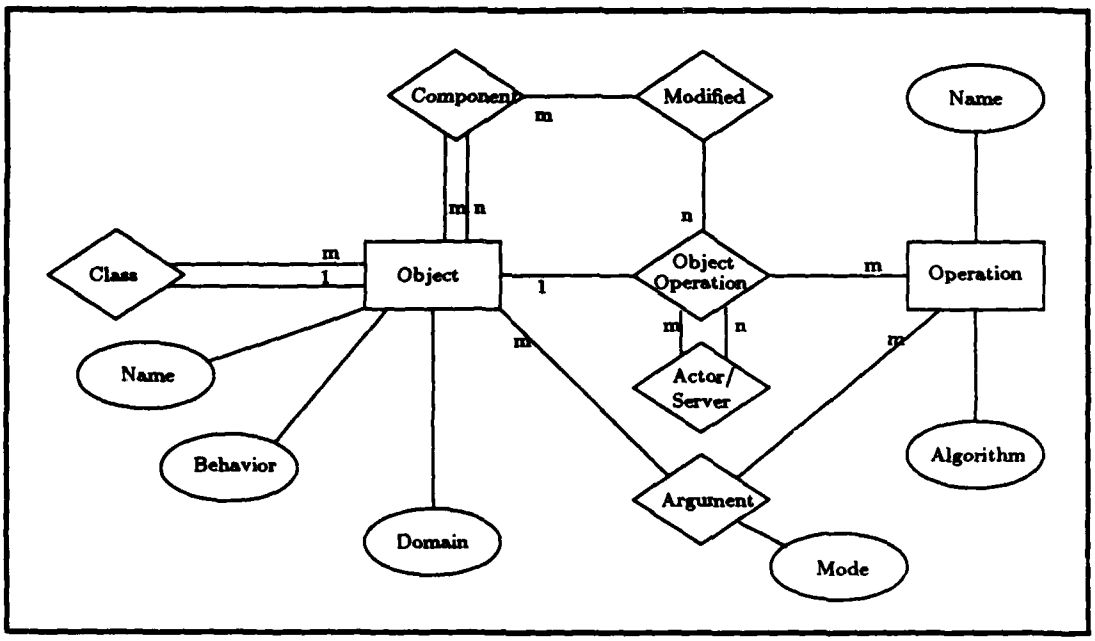


Figure 4.8. An E-R Diagram for the Object Model

The modelbase elements: the object model, the methodology steps, the design heuristics, and the hook book all require text or graphic tools for editing and manipulating the images supported by the database component.

- The *modelbase component* will provide editors for the text and graphics databases required by the methodology.

Objects		
name	behavior	domain

Operations		
object	name	algorithm

Component Objects	
parent	child

Argument Objects		
object	operation	mode

Modifies Objects	
operation	component

Class Objects	
class	object

Actor/Server Objects	
actor	server

Figure 4.9. Relations for the Object Model

V. Prototype Implementation and Evaluation

5.1 Introduction

In the previous chapter we presented the overall objectives, general capabilities, specific capabilities, and the features required of a *Decision Aid for Object-Oriented Design*. We used the feature chart (Figure 4.1) and storyboards (Figures 4.2–4.7) to represent the top level design of the dialogue component, and discussed initial design of the database and modelbase. We now turn to the implementation and evaluation of a prototype, which can be used to evaluate the methodology and concepts presented earlier.

We start by discussing our determination of the hardware and software tools used to implement the dialogue, database, and modelbase components. Then we describe the implementation of the selected components. We conclude with an evaluation based on key decision points relating to the prototype's development.

Case studies of DSS usage show that "Key factors explaining successful development are a flexible design and architecture that permit fast modification and a phased approach to implementation" [44]. Thus though we've stated the initial requirements and design somewhat formally, only time will tell whether or not the ensuing system will be accepted and of value to the users. Consequently, we took a phased approach to implementation, and evaluation which would allow user response and feedback before investing in further development.

The first phase, implemented as a part of this thesis, was to implement the storyboards described in the previous chapter. Only as much functionality as was necessary to demonstrate the potential value of the methodology was implemented. However, even this first kernel system must be "a small but usable system to assist the decision maker" [44]. Additional work is recommended in the final chapter as an area for future study and research.

5.2 Hardware and Software Selection

The primary hardware available for prototype development included the Engineering Department's VAX 11/780, Sun workstations, and Zenith Z-248 micro-computers. The following discussion relates the prototype's components to the specific hardware and software used to support them.

5.2.1 Dialogue

To support a rapid prototyping or experimental development approach to implementing the storyboards, a software environment was needed which would provide the following capabilities:

1. It must be able to provide *representations* of the storyboards as screen displays through which users can access the various required functions.
2. It must support a variety of *operations* including accessing text and graphics files, entering text, drawing graphic representations, and selecting from menus.
3. It must support display of context sensitive help and methodology information as *memory aids* to the user upon request, without disturbing other work in progress.
4. It must provide *control* over the database and model base storage and retrieval as well as between the various storyboard functions.
5. It must provide a high level of interactive programming such that changes may be made rapidly and development time is minimized.
6. It must be readily available for use on available hardware.

Based on these requirements, we initially considered the Sunview environment on the Sun workstations. However, the graphics capabilities seemed too low a level to be used for a rapid prototyping approach. In addition, the limited availability of Sun workstations led us to look for a software environment that would run on the more readily available Zenith Z-248 micro-computer.

Since much of the research for this thesis involved analysis of object-oriented principles, we supposed an object-oriented programming environment might well suit our needs. The Smalltalk/V Object-Oriented Programming System (OOPS) [17] was

obtained and installed on a Zenith Z-248 micro- computer and evaluated as to its capabilities for implementing the prototype. Smalltalk/V seemed to provide a rich toolset supporting software reuse, bitmapped graphics, and interpretive compilation which would enhance rapid development of the prototype.

5.2.2 Database

In Chapter IV we said the database must support the requirements documents, design tools, and the object model. We use that framework to discuss database tool selection and implementation.

5.2.2.1 The Requirements Database. The DCDS Support System provided a database for developing a requirements specification including a data dictionary and graphical representation. Since a major objective of the thesis was that a design support tool should integrate with other development tools, we felt it imperative to use an existing system to support requirements analysis. Another factor in selecting the DCDS Support System was its applicability to developing the kind of real-time distributed systems for which object-oriented design seems to be best suited.

5.2.2.2 The Tools Database. We expected Smalltalk/V to provide complete support for reading and writing text files and saving graphics images. Organization of internal data would also be provided from within the programming environment.

5.2.2.3 The Object Model Design Database. Initially we wanted to implement the object model as a new language extension of the DCDS. The DCDS Support System's entry feature made it quite easy to define the new language, but the constraints on DCDS access from within Smalltalk precluded implementation. The alternative was to use Smalltalk/V data structures to implement the object model.

5.2.3 Modelbase

The help and heuristics files comprising the modelbase were implemented as text files.

5.3 Prototype Implementation

5.3.1 Dialogue

Implementation of the dialogue component was primarily a matter of writing Smalltalk programs to display and provide functionality to the storyboards. Generally following our own OOD methodology, the high level design can be seen directly in the feature chart of Figure 4.1. Restructuring the feature chart into an OOD block diagram provides the view shown in Figure 5.1.

From the block diagram, we developed the initial detail diagram based on the functionality shown in the feature chart. The initial detail diagram was modified as it became clear which objects would become separate modules and the final diagram is shown in Figure 5.2. We then proceeded with implementation of system functions using an incremental approach as follows:

1. Implement the storyboards as windows with the following general format: a top or title pane, a features pane, an objects pane, and a text pane. Since each of the storyboards had this general format, we developed a storyboard **class** and made each of the five main storyboards a subclass, thus inheriting the basic attributes of the class.

2. Implement the main menu which would allow movement between storyboards and access to help and hook book features. The main menu would be the driving object and keep track of the decision aid's state.

3. Implement the features panes by providing the list of features for each storyboard, and the mechanism to display the appropriate list of objects for the selected feature in the objects pane. The mechanism for displaying text in the text

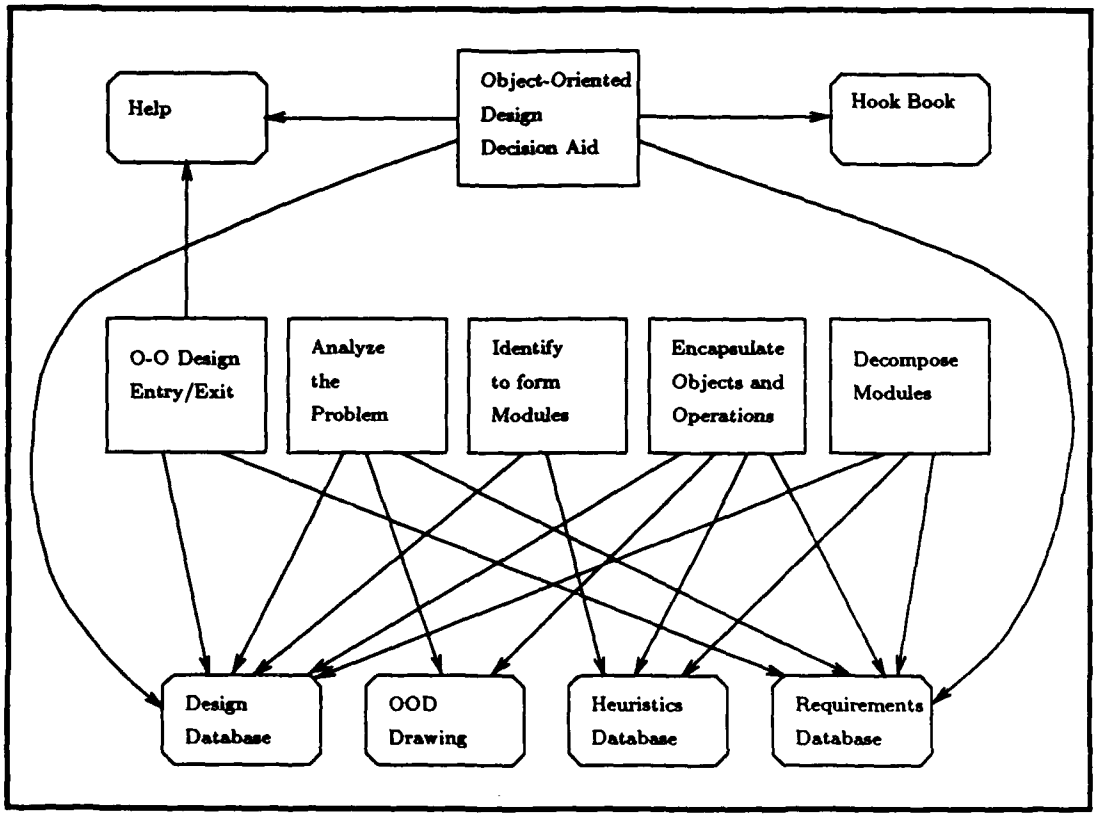


Figure 5.1. Block Diagram for the OOD Decision Aid

pane was also implemented. As individual features were added from here on, subclass specific modifications to the general control mechanisms would be added.

4. Implement the hook book as a browser which would be a separately displayable and controllable window. Early implementation of the hook books functions allowed its use for making notes regarding further development.

5. Implement the design database as a separate abstract data type. Representation of the database changed several times. Initially it was a complex hierarchical data structure using recursive algorithms for accessing its components. Eventually it was changed to the simple set of relations described in Chapter IV. As an abstract data type, the changes were confined to the database object itself. Initially, opera-

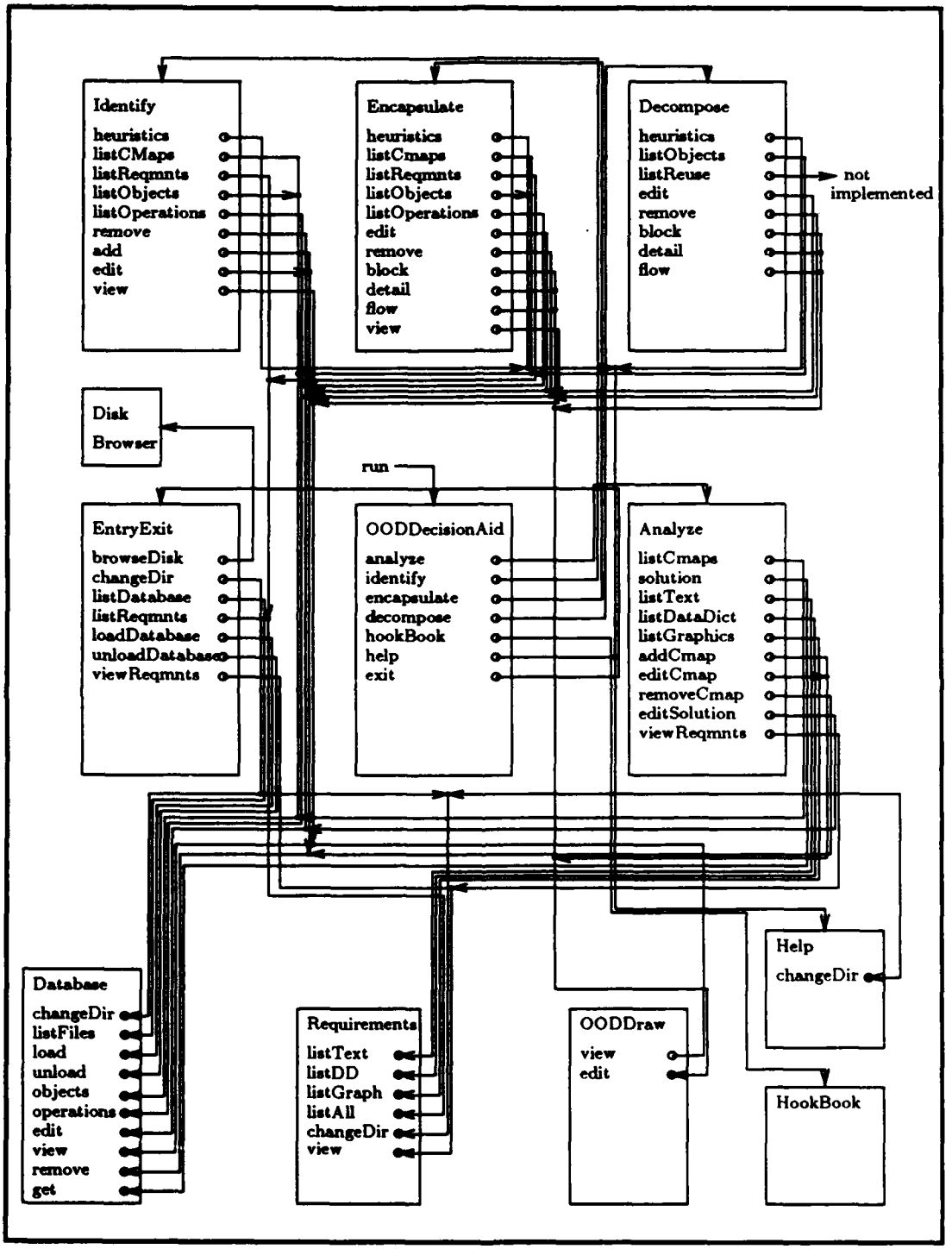


Figure 5.2. Detail Design Chart for the OOD Decision Aid

tions were implemented to load and unload the database from disk. Operations were added later as they were required by other storyboard features.

6. Implement access to the requirements database. This began as merely accessing text files as needed. But as redundancy began to appear in several storyboards, the requirements database became a separate object with its own operations and window for displaying text. As previously mentioned, graphics files were displayed through a DOS shell to the VTEK PLAY program with automatic return to Smalltalk/V.

7. Implement the design graphics capability. A subclass of the free drawing application provided with Smalltalk/V was created to provide the specific requirements of the decision aid. Rather than limit the features for each application to the subset required, other useful features were left for further experimentation with graphics capabilities. The single drawing object developed could be used for both concept maps, and the three types of design graphs required.

8. Implement the help feature. Initially, help was forced to the main text pane, but it soon became evident that that might destroy current work in progress. So the help feature was also made a separate object given browser capabilities similar to the Hook Book. Additionally, users were given the ability to modify and thus tailor the help information to their own needs. Since help was to be context sensitive, each storyboard would maintain a set of help file names for access only by the help object for that storyboard.

9. Implement the heuristics display capability. Since heuristics text files needed to be accessible by topic, as with help, it became evident that this was just another form of user help and so we implemented it as an instance. Being context sensitive, a separate list of heuristic file names would be maintained by each applicable storyboard.

10. Implement a database browser to provide a window on the database for adding, editing, and removing relations. This kept window operations separate from more general database access operations.

As each decision aid feature was implemented, the controls to access that feature were put into the appropriate storyboards and tested. In this way we always had a working model from which we could test new features. This method of incremental development worked very well for single user development and facilitated experimentation with various design ideas. It also made possible rapid modifications to the software based on hook book entries.

5.3.2 Database

5.3.2.1 The Requirements Database. The critical task involving implementation of the DCDS was to be able to access its data from within the dialogue.

Our first alternative was to use the PC version of the DCDS to provide direct access to the database from within Smalltalk/V. However, the PC version was no longer supported by the vendor and we could not implement it successfully on the Zenith Z-248. Our second option was to use Smalltalk/V's communications package to directly access the VAX VMS version of the DCDS. The extremely slow response time for the VAX version, combined with the requirement of using a Textronix 4105 emulator to access DCDS graphics eliminated this alternative.

Our final choice was to download DCDS requirements data and access it off-line. The data dictionary information was listed to text files through the DCDS Support System's query function and graphics screen displays were captured through Scientific Endeavors Corporation's VTEK Textronix terminal emulator. Smalltalk/V was able to input the text files directly and the graphics images were displayed through a DOS shell escape and execution of VTEK's PLAY program.

5.3.2.2 The Tools Database. Implementation of Smalltalk/V on the Z-248 made available the following software development tools:

1. a class browser for accessing source code;
2. a disk browser for accessing text files on disk;
3. a text editor providing screen editing, search, search and replace, saving changes, cut and paste between windows, and various other helpful features;
4. a free drawing editor for developing graphic images;
5. a DOS shell for exiting to DOS and executing external programs;
6. access to the entire system source code for use as reusable components or templates;
7. a project manager for controlling code changes having to do with the application;
8. an object unloader for loading/unloading the design database;
9. a complete multiple window and mouse driven environment for maintaining multiple simultaneous views of the software in development;
10. incremental automatic compilation upon saving source code;
11. a debugger providing a walkback feature for tracing errors in execution.

5.3.2.3 The Object Model Database. Implementation of the object model in Smalltalk consisted of declaring a new class and selecting the data structures to represent the model. Initially, a hierarchical model was developed using a directory to contain the various attributes of each object. This soon became complicated and cumbersome and a simpler relational approach was taken, more directly implementing the relations of Figure 4.9.

The database was defined as a dictionary of relations with the name of the relation as the key to the dictionary. Each relation was then implemented as an ordered collection of arrays with each element of the array containing a string item or a pointer to an object or operation in the appropriate relation. All these relations were objects in the database dictionary with the relation names as the keys. Figure 5.3 graphically depicts this structure.

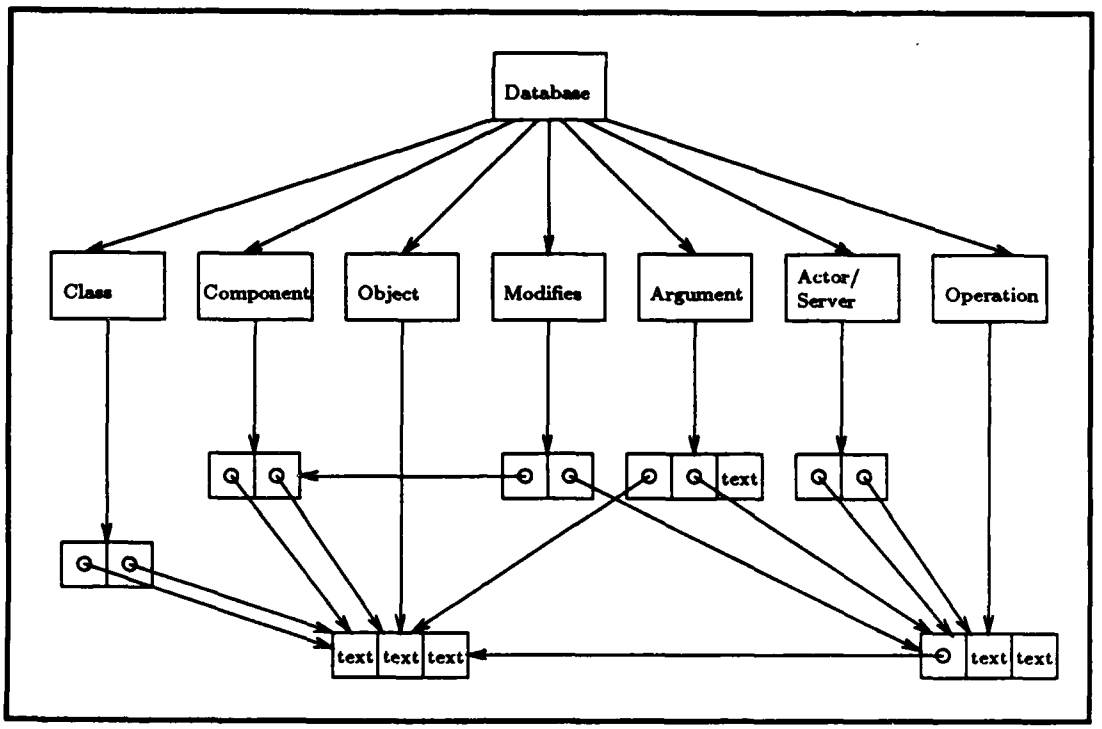


Figure 5.3. OOD Database Internal Structure

The standard data structure operations were implemented to hide this structure from the using storyboard features. Additional special operations were then added to support unique database accessing requirements to simplify code in the storyboard operations.

5.3.3 Modelbase

Implementing the modelbase was a simple conversion of the heuristics and methodology instructions from this document to the text files which could be referenced by the help and heuristics features. These files were then added to the default lists of help and heuristics files built into the storyboards. The OODDecisionAid controller module maintains the list of help files and these were hard coded as defaults which could then be edited, removed, or added to by the users. Each storyboard

contains its own list of heuristic files which was also pre-loaded with defaults from the examples provided in Chapter III.

5.4 Evaluation of the OOD Decision Aid

Due to the required flexibility and objectives of the prototype development effort, Riedel and Pitz's Utilization Shaped Evaluation (USE) model was chosen as the most effective means of evaluation for the OOD Decision Aid. As we said in Chapter II, the USE model focuses on what *use* will be made of evaluation information.

First to be considered were what decisions were to be made regarding development, who would make those decisions, and what were the critical windows of decision opportunity. Second, the mission the DSS is to accomplish was defined, along with its primary users and environment. One or more methods and associated measures were proposed for each decision point identified. Finally, an evaluation was performed to answer the evaluation questions. This section describes the results of each of these four steps and presents our conclusions from the evaluation.

5.4.1 Decisions

Table 5.1 lists the general decisions which need to be made throughout the estimated life cycle of the target system. As a thesis project, this life cycle extends only through the implementation and evaluation of a prototype. Also, the evaluation criteria must concern itself with additional questions regarding basis for future research and qualitative significance of the proposed thesis.

5.4.2 The Mission, Users, and Environment

To determine the decision aid's effectiveness, the system must be evaluated in terms of its mission objectives and its appropriateness for the user and the operational environment. The following discussion defines these three elements.

Table 5.1. Life Cycle Evaluation

Decision	*Decision Maker*	*Phase*
1. Does the concept map accurately reflect the relationship between the decision points in the process?	user	RD[a]
2. Do the story-boards accurately reflect an algorithm capable of aiding the decision-making process?	user/ designer	RD/RS[b]
3. Is the kernel appropriate for development of a prototype?	designer	RA[c]
4. Is the prototype feasible with available technology?	designer	RA/PD[d]
5. What components will be used to build the prototype?	builder	PD/DD[e]
6. Is the completed prototype technically correct?	builder/ designer	PI[f]/VV[g]
7. Does the prototype accomplish the user's objectives?	designer	VV
8. Is the user satisfied with the prototype's usability?	user	CD[h]
9. Does the expected value of the DSS justify further development?	policy maker	CD
10. Does the approach taken represent a significant contribution to the engineering community?	policy maker	CD

[a] Requirements Determination
[b] Requirements Specification
[c] Requirements Analysis
[d] Preliminary Design
[e] Detail Design
[f] Prototype Implementation
[g] Validation and Verification
[h] Continued System Development

5.4.2.1 Mission. The goal of the Object-Oriented Design Decision Aid is to improve the timeliness and quality of design decisions made by software developers in producing an object-oriented software design specification. The OOD Decision Aid will present both methodological and qualitative information to the designer as needed and in a manner that enhances the object-oriented design process. The aid will also guide the designer in a structured fashion through the four decision phases of object-oriented design: analysis, identification, encapsulation, and decomposition. These phases deal with the following questions:

1. What is the problem and what strategy is proposed for solving it?
2. What are the abstract objects and operations of interest?
3. How should objects and operations be associated in modules and what are the interfaces between them?
4. Should a module be further decomposed or may it be constructed from known components?

To support this broad goal, the prototype must meet the following three objectives as stated in the *Scope* section of Chapter I. They are as follows:

1. It must emphasize the four decision steps of analysis, identification, encapsulation,and decomposition.
2. It must demonstrate the benefits of on-line access to requirements specification textual, data dictionary, and graphical information.
3. It must provide a user interface which may be easily adapted by the user—even to the extent of altering the methodology itself.

5.4.2.2 User. This section might be more appropriately labeled *users* since several users are involved in making decisions regarding the development and use of the DSS. As noted in the section on mission, the end user or person who works directly with the aid will be a software engineer, programmer, or designer. But the evaluation must also address decisions made by policy makers and program managers. The program manager is the one most likely to sponsor development or

procurement of the DSS and would be in direct management of its implementation in the software development environment. The policy maker would be at the approval level for the DSS and have overall control of systems development beyond the target project or system. In the case of this thesis effort, the policy maker will consist of a thesis advisor and the members of the thesis committee.

5.4.2.3 Environment. The decision aid is to be placed in a software development environment consisting of a number of tools and resources already in place. Its task will be to provide a framework for integration of those tools, resources, and information bases already in existence which support the OOD process. As a highly adaptive environment, subject to rapid technological change, the system must be able to readily integrate new tools and resources in support of a variety of software development projects.

5.4.3 Choice of Evaluation Methodology

Considering the factors stated above and the expected questions to be raised throughout the life cycle, methodologies were chosen to evaluate each question stated in Table 5.1. For each question and methodology, the proposed general measures of effectiveness are given in Table 5.2.

5.4.4 Evaluation Results

The following discussion of the results of evaluating the developed prototype follows the format established by the ten questions outlined in Table 5.1. Each question's answer provides a summary of the results of the evaluation as well as a discussion of the method and measures used.

Table 5.2. Evaluation Methods and Measures

Decision	*Methodology*	*Measure*
1.	Attitude Survey	Sample of Experts Subjective Rating
2.	Process Evaluation	Subjective Rating
3.	Feature Analysis	Sufficiency
4.	Systems Analysis Value Analysis	Available Methods DSS Costs
5.	Cost/Benefit	Component Cost
6.	Systems Analysis	Verification
7.	Systems Analysis Attitude Survey	Verification Subjective Rating
8.	Human Factors	Subjective Rating
9.	Rating and Weighting	Subjective Rating
10.	Rating and Weighting	Subjective Rating

5.4.4.1 Does the concept map accurately reflect the relationship between the decision points in the OOD process?

The OOD methodology concept map was initially developed from the literature and interviews with three professors in software engineering and object-oriented development from the institute's School of Engineering Department of Electrical and Computer Engineering and Department of Mathematics and Computer Sciences. After a composite concept map was developed, it was provided to the original three experts, plus four more for further evaluation. Their comments were used to produce

the final version which was then accepted by all six reviewers. The resulting concept map was used to develop the methodology and was shown in Figure 3.1.

The experts were asked whether they strongly agreed, agreed, disagreed, or strongly disagreed with the proposition put forth by the question. On a four point scale, the average response was *insert numerical rating here* with none disagreeing. We conclude then, based on expert opinion, that the concept map does accurately reflect the relationships between the key decisions which must be made in the object-oriented design process.

5.4.4.2 Do the story-boards accurately reflect an algorithm capable of aiding the decision-making process?

Answering this question required analysis of the OOD process. The process evaluation method used required linking processes with their desired outcomes. Accepting the methodology described in Chapter III as an accurate reflection of the OOD process, we only needed to link the functions reflected in the storyboards with the steps required by the methodology. In other words, we assumed following the steps of the methodology would result in the desired outcomes.

By observation, the storyboards directly embody the step- by-step execution of the proposed OOD methodology. Each step in the methodology is directly supported by a single window which provides the required functionality of that step. We conclude then that the storyboards reflect the algorithm presented in the methodology.

The additional question of to what degree the methodology itself represents an algorithm beneficial in aiding object-oriented development is addressed in the evaluation of the methodology presented in Chapter III.

5.4.4.3 Is the kernel appropriate for development of a prototype?

What we needed to determine is if the kernel represented a sufficient set of features such that a system implementing those features would present a product

useful for helping the user make decisions. We show here how the decision aid supports each step of the methodology.

Analysis Provide the user with access to the three requirements specification views of graphics, text and data dictionary for use in deciding the scope and solution to the problem.

Identification Provide the ability to view requirements, the solution strategy, and concept map for identifying an initial list of objects and operations. Also provide access to software engineering concepts and heuristics for identifying objects and operations.

Encapsulation Provide a template for describing objects and operations such that the relationships among them indicate how they might best be encapsulated into modules. Also provide access to software engineering principles and measures to help determine the quality of the resulting modules.

Decomposition Provide access to a set of reusable design modules and the ability to further describe module functionality as an aid to determining whether further decomposition is required.

We conclude that the features just described as provided by the storyboards should provide significant help to the user for making design decisions. The additional features supporting maintaining the object model text and graphic representations also provide the ability to capture those design decisions once they are made.

5.4.4.4 Is the prototype feasible with available technology?

This question was answered in detail in the implementation section of this chapter. While the technology to implement any of the specified features clearly exists, the question became more one of availability, appropriateness, and manpower costs considering the prototyping task to be performed. Analysis of the features required of the prototype, and of the resources in both time and availability of hardware and software, narrowed our choices to a minimally implemented Sun workstation based system, or a more fully developed prototype on a Zenith Z-248. We leave further discussion as to selection of the hardware and software to the next section.

5.4.4.5 What components will be used to build the prototype?

The cost/benefit analysis was a constraints based approach due to the expected availability of development resources. Constraints primarily involved manpower costs (time). Constraints required that the prototype be completed within 60 days of the completed initial specification using one person relatively unfamiliar with the software development tools to be used.

Building the kernel required a window based user interface providing powerful programming tools for rapid experimental development. The only initially available tool was the Sunview environment on AFIT's Sun workstations. While the Sun would provide a more powerful workstation level environment, its inability to access the requirements database graphics and the expected learning curve of the complex set of fairly low level tools would have limited the breadth of the functions implemented in the prototype.

As an alternative, we found the Smalltalk/V object-oriented programming environment adequately met the need for functionality, a powerful and high level tool set, and at minimal cost. Installed on an IBM-PC/AT class Zenith Z-248 micro computer, we had full off-line access to the requirements database. Without the availability of this environment, the functional requirements for the kernel would have to have been greatly reduced.

5.4.4.6 Is the completed prototype technically correct?

Testing of the software for the prototype was somewhat informal and followed the incremental development approach discussed earlier in this chapter. Functions were tested for technical correctness as they were added to the prototype. Walk-throughs were accomplished periodically testing each of the features specified in the description of the requirements in Chapter IV.

No attempt was made to formally evaluate the performance characteristics of the prototype since feature functionality rather than performance was the objective.

However, it was observed that Smalltalk/V cannot support very large applications without having to continually swap objects in and out of memory. This object swapping considerably reduces response time. While this problem was alleviated somewhat through use of a RAM disk, an operational system would require either a workstation version of Smalltalk, or possibly the 286 version running in protected mode and using at least a full megabyte of RAM.

5.4.4.7 Does the prototype accomplish the user's objectives?

The objectives of the prototype were stated in Section 5.4.2.1. We can see that the first objective is met since the prototype encompasses all four decision steps and implements the OOD model in a database. The second objective is met through providing access to requirements specification data from the appropriate storyboards. Finally, the third is met through implementation of the help and heuristics features. Since the user can edit, add, or remove heuristics and help files, the details of the design methodology can be easily altered.

The selection of the Smalltalk/V OOPS as a development environment provided an additional level of adaptability since the users themselves are expected to be software engineers. Smalltalk provides an easy to use programming environment rich with development tools. Such an environment should make it very easy for a user to customize or extend the decision aid.

5.4.4.8 Is the user satisfied with the prototype's usability?

A group of sample users from a graduate level advanced software engineering class was given an hour to play with the user interface of the prototype. They were then asked to answer the following questions.

1. Rate the content of the information displayed. That is, how well did the system keep you informed of where you are or what you are doing.
   insufficient 1 2 3 4 5 6 7 sufficient

2. Rate the methods used to communicate with the user.
   Inputs:
   insufficient 1 2 3 4 5 6 7 sufficient
   Outputs:
   insufficient 1 2 3 4 5 6 7 sufficient
3. Rate how well you feel the system produced user induced errors.
   insufficient 1 2 3 4 5 6 7 sufficient
4. Rate how well you feel the system allowed you to recover from user induced errors.
   insufficient 1 2 3 4 5 6 7 sufficient
5. Rate your expected ease of learning for the system.
   difficult 1 2 3 4 5 6 7 easy
6. Rate the ease of use of the system.
   difficult 1 2 3 4 5 6 7 easy
7. Rate your ability to direct or control the activities of the system.
   insufficient 1 2 3 4 5 6 7 sufficient
8. Rate your overall satisfaction with the interface.
   unsatisfied 1 2 3 4 5 6 7 satisfied

The results of that evaluation are displayed in Figure 5.4 and Figure 5.5. The first graph shows the mean answer by question on a rating scale of one to seven. The graph shows an average standard deviation per question of 1.25. Such a high standard error made it very difficult to draw a firm conclusion as to the users satisfaction. We could only say that we are reasonably confidant that the user is more satisfied than not satisfied. This statement is based on the hypothesis that the user's rating is greater than the scale's midpoint of 3.5, tested at the .05 level.

The second graph was developed to try to answer the reason for the high standard error. A low standard deviation here probably indicates a less well considered answer to specific questions. The graph shows students generally answered each question relative to their overall impression rather than to the scale. Answers also reflect the impact of varied backgrounds or preferences of the users.

This aspect of the evaluation proved to be less useful than hoped. What is needed is a more thorough evaluation by users familiar with the concepts involving

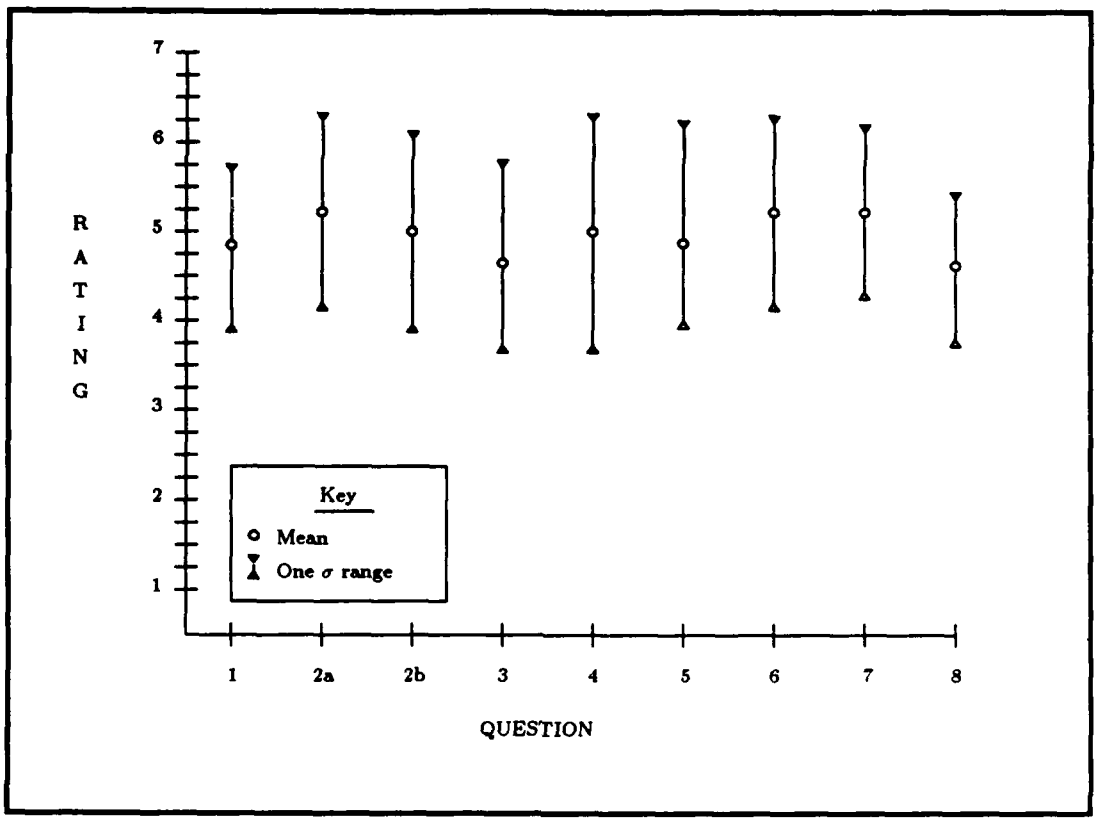


Figure 5.4. User Interface Evaluation by Question

specific comments rather than a numerical rating. The results in this case do not even allow us to identify specific areas requiring improvement.

5.4.4.9 Does the expected value of the DSS justify further development?

The finished prototype was demonstrated for the thesis committee which was asked to provide a subjective evaluation of the prototype regarding justification for further research and/or development. Their comments are summarized as follows:

- The methodology requires further development before consideration of tool development is warranted.
- The Smalltalk interface is very nice, but we're not convinced it can support the amount of data which would need to be processed in even a medium size project.

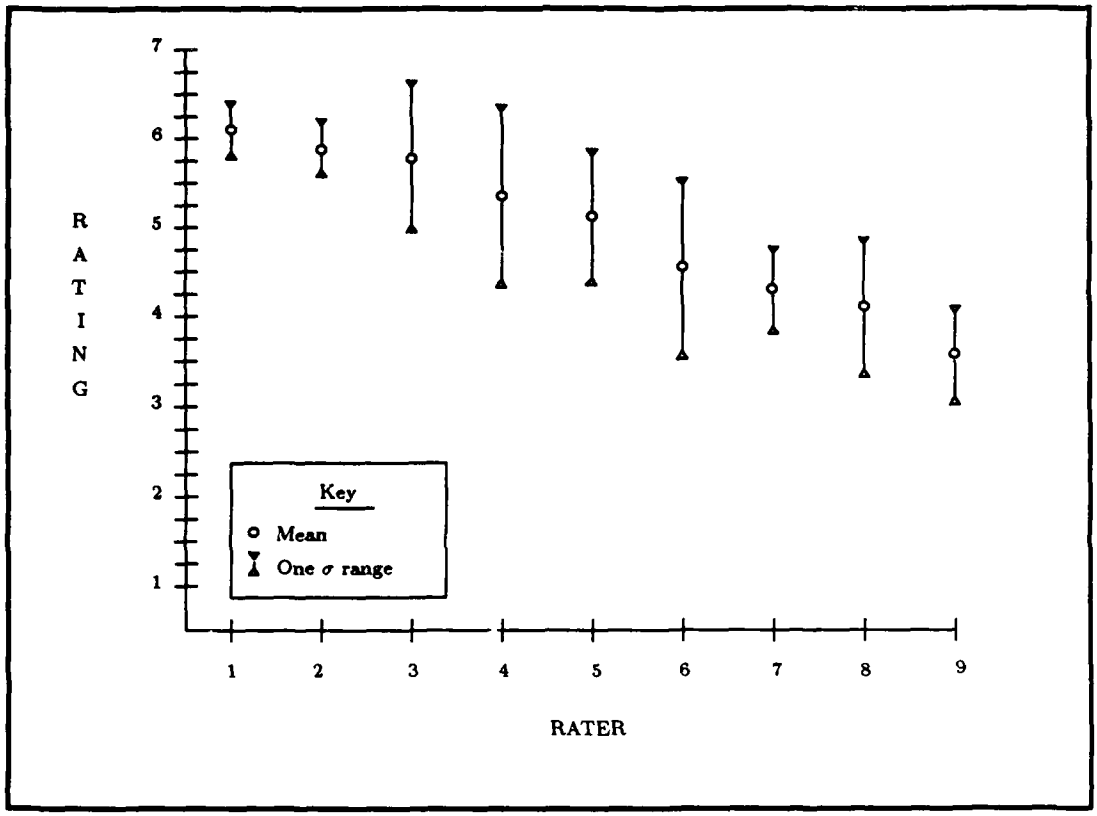


Figure 5.5. User Interface Evaluation by Rater

- The concept mapping idea will be used as the basis for further research in object-oriented analysis.

Two schools of thought come into play in this analysis. The first says we shouldn't build anything until we know exactly what we want. This is the traditional approach and it makes sense considering the high cost of full scale development. The problem is that in some cases, we really don't know exactly what we want. Such is the case with most decision support systems. The adaptive design approach lets us experiment with ideas until we either discover or recognize from experimental results what it is we are really looking for. This thesis took that second approach. While the prototype is insufficient for problems of any scale, it should be sufficient for experimenting with the OOD methodology.

5.4.4.10 Does the approach taken represent a significant contribution to the engineering community?

The thesis committee was also asked to provide a subjective response regarding the contribution of this research to the engineering community. The idea here is to determine whether or not the application of DSS concepts to developing software support tools and environments is unique, and whether it is of sufficient value to be of interest to other software engineers. The committee's response is somewhat qualified. Whether this approach is new, or just the same thing we've been doing—but with another name—is the argument DSS adherents have been trying to deal with for years. As to the potential interest to others in the field, the answer is clearly yes! The ideas demonstrated by this prototype should stimulate others to pursue research in the application of DSS and OOD concepts.

5.4.5 Conclusions from the Evaluation

The primary conclusion we derived from development and evaluation of the prototype decision aid was regarding the application of the decision support system concepts. Using the adaptive design approach, along with the techniques of concept mapping, storyboarding, and the ROMC model, we were able to develop a prototype which more than met all of the stated objectives. In addition, development time was held to less than two months, cost was minimal, and the final product was evaluated as a successful implementation of our OOD methodology and a valuable tool for further research.

An unexpected benefit resulted from the opportunity to use the Smalltalk/V OOPS. Its powerful set of tools and reusable software components made the programming task relatively simple as compared to similar systems developed using a more general purpose language without an extensive support environment. Further comments and recommendations are provided in the final chapter of the thesis.

VI. Conclusions and Recommendations

We conclude this thesis with a summary of the work accomplished and how it related to the specific objectives for the study. Next we present conclusions drawn from the effort and its results. Finally we include recommendations for continued research regarding the methodology and development of the decision aid.

6.1 Summary

We began this effort with the primary objective of developing an object-oriented design methodology which would support transition from a non-object-oriented requirements specification. We also wanted to implement that methodology in a tool which could be used for object-oriented development and research. But how does one go about coming up with a new methodology? That is where decision support techniques began to play a role.

The concept mapping technique allowed us to more fully understand the object-oriented paradigm and the design process itself. Creating and analyzing the concept maps, and various discussions as to the definition and application of DSS led us to propose that software design fit the definition of semi-structured decision processes described in the DSS literature. Following that lead as a means of developing the methodology, we continued the process of concept mapping numerous object-oriented development related sources to finally arrive at the concept map of Figure 3.1.

Once the key decisions were identified, describing the steps of the methodology was a matter of determining what help could be provided the designer to aid the decision making process. This entailed concept mapping the requirments specification methods and consolidating them to form a generic requirements specification model. Having done that, we still needed to have as a basis a representation of the object model. Neither the theoretical nor the programming language-oriented model proved quite adequate for design. Using the adaptive design methodology, we added

features to the theoretical model as necessary to provide just enough constraints to formulate a workable model. We then evaluated the finished methodology with a sample problem and it seems to have produced a sufficient design specification for implementation.

Our efforts then turned to determining the requirements for a tool. Again the decision support concepts came into play as we used the feature chart and story-boarding techniques to specify the requirements for a decision aid. We also used a DSS evaluation technique to establish life-cycle measures for development of a prototype. Several weeks were spent in evaluation determining the appropriate hardware and software configuration for implementing the prototype. Then, about seven weeks were spent developing the software for the decision aid.

Finally the prototype was complete and the final evaluation steps were taken. The prototype more than adequately provided a test-bed for further research and evaluation of the methodology, as well as an example of the application of DSS techniques to this type of tool. Unfortunately we were unable to more thoroughly evaluate the methodology through controlled experiments with the tool.

Our stated goals were to address the problems of *transition*, *integration*, and *adaptation* as they apply to current development of support environments for object-oriented development. We developed a methodology for OOD which transcended the limitations usually placed on requirements specification techniques and programming languages. In addition, we implemented that methodology in an environment which included direct access to a multi-view requirements specification, and provided a user adaptive interface. Additionally, we based our methodology on a unique approach that promises to help designers make decisions, not just capture decisions once they're made. We feel the results show we've not only met but exceeded our original goals.

6.2 Conclusions

As stated in the evaluation of the prototype, we feel our most significant conclusion from this study has been the applicability of DSS concepts to the development of software environments. Evaluation of eight software development methodologies or tools showed no evidence that software environment developers are addressing techniques that will help the user make good design decisions. We believe drawing complicated graphics or following rigorous documentation techniques will not greatly improve the state of current programming practice until we provide the designers and programmers the on-line tools to help them make better design decisions.

A second conclusion is regarding the use of the concept map. We found the concept map to be an excellent informal tool for communicating understanding. As such, we feel it may lead to a better means of representing the user's view of the problem than many formal specification methods which are often incomprehensible to the user. We also feel the method presented for using concept maps to describe the solution strategy is more descriptive and may lead more directly to a set of candidate objects and operations than the textual paragraph of Abbott, Booch and others. We caution, however, that the inherent value of the concept map is in its simplicity and informality. Over formalizing and constraining its use may have a corresponding negative effect on its ability to communicate understanding.

An observation we made during the course of the research is that methodologies which are language independent seem to have the most chance of surviving and being used over the long haul. While language specific features may certainly be very valuable in a given implementation, too many tools embed such features in the very essense of the methodology. We found many of the tools with embedded language features were simply not being widely used as commpared to language independent methods such as Structured Analysis and Design. A corollary to this observation is that the more complex the method, the less it seems to be used.

A final conclusion was drawn from using the rich tool-set and powerful reusable components available with the Smalltalk/V OOPS. We feel that an OOP environment might be very successfully used as an interface for developing software support environments for more traditional target languages such as Ada. While this suggestion is contrary to the Stoneman document [18], that document was written nearly a decade ago and may not provide the best solution for developing APSEs. Another application in which Smalltalk should prove beneficial is in the development of decision support systems in general. The experimental programming approach for which Smalltalk was developed seems well suited for adaptive design of DSS.

6.3 Recommendations

We hope to continue research into the application of DSS concepts to development of environments to support traditional software design. In fact, we think further research would show that virtually all design requires semi-structured decisions which may well be supported by DSS. To our knowledge, DSS concepts have not been applied to such applications and research in that direction is clearly needed.

We also recommend continued development of the Decision Aid for Object-Oriented Design as a promising method for exploring research into object-oriented development methodologies. We suggest development continue on a higher performance system such as a work-station or a non-DOS PC environment without the suffocating 640 kilobyte memory limitation. The Smalltalk/V 286 version runs in protected mode and may provide an excellent alternative at minimal cost.

The methodology we developed is only a starting place. We would like to see an extensive evaluation accomplished using the OOD Decision Aid as a test bed. A particular area of concern is in decomposition of modules and application of inheritance and reusable components.

Finally, we recommend further research into the use of concept maps as a tool for communicating understanding. An interesting approach based on their use in requirements determination may be to use concept maps as a basis for an object-oriented approach to the entire development lifecycle.

6.4 Closing Remarks

This thesis effort was a success in the sense that it demonstrated a unique approach in the application of decision support systems concepts toward developing tools and techniques for software support. By approaching formation of the methodology and tool from the users' point of view and the decisions which they must make, the object-oriented design methodology was presented as a technique which should help in the design of reliable maintainable software. OOD was shown as a benneficial addition rather than as a threat of drastic change to the software development environment. Presented as such, OOD should have a much greater chance of being accepted and used by the software development community.

Appendix A. *Executive Summary*

A Decision-Based Methodology for Object-Oriented Design

A Decision-Based Methodology for Object-Oriented Design

Captain Patrick D. Barnes and Dr. Thomas C. Hartrum
Department of Electrical and Computer Engineering
Air Force Institute of Technology

December 16, 1988

Abstract

The task of object-oriented development raises a new set of design problems. Addressing the decisions which must be made in applying object-oriented principles to design is the focus of this paper. A structural object model is presented and the concepts of decision support systems (DSS) are applied to the formulation of a decision-based methodology for object-oriented design. An overview of the development of a decision aid for evolution of the methodology is also given.

Contents

1 Introduction

Escalation of software development and maintenance costs as well as demand for software solutions to increasingly complex problems have mandated new techniques for engineering reliable, maintainable computer software. One approach to improving software quality is the use of the *object-oriented* paradigm for design. Pressman [37] lists desirable software engineering principles specifically addressed by OOD as abstraction, information hiding, and modularity. While the principles themselves are not new, he states, "only OOD provides a mechanism that enables the designer to achieve all three without complexity or compromise."

But there is more to design than the paradigm we choose for structuring, conceptualizing, or representing a system. The design process can be seen as combining

> ... intuition and judgement based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation [37].

This description indicates that a software design environment must support judgment and choice, embody design principles and/or heuristics, guide an iterative development process, and enable qualitative evaluation of the finished product. While several methodologies have been proposed for an object-oriented approach to design[1], they seem to focus primarily on the representation of the design rather than the process.

This paper presents an approach to developing an object-oriented design methodology based on the concepts of decision support systems. The OOD process is not redefined; rather it is stated in terms of the decisions a designer must make while accomplishing OOD tasks. First, a general object-oriented model for design is presented. The decisions involved in OOD are then stated and a methodology is elaborated based on those decisions. Finally, an overview of the first stage development of a decision aid is discussed.

2 An Object Model for Design

2.1 Defining the Model

Two models of the object-oriented paradigm were analyzed for application to design. The first was a theoretical model [10] based on objects, behaviors, and attributes. The second was the Smalltalk language-based model [17] which adds class, inheritance, messages, and methods. The theoretical model proved to be too ambiguous to rigorously depict relationships between objects, and the OOP model too restrictive to implementation constructs. A new model which would meet the needs of design was needed. Such a

[1] Examples of OOD support tools are [19], [14], [11], [12], [42], and [2].

model was derived by beginning with the more abstract theoretical model and adding refinements derived from the Smalltalk experience to solve design related problems.

The resulting object model is pictured in the concept map of Figure 1 in the appendix and is formally defined as follows:

object A unique entity defined by *attributes* which serve to identify the object, and *relations* which associate it with other objects, *relations*, and *operations*.

operation The description of how an object performs some *behavior*. As with objects, *attributes* serve to identify the operation and *relations* associate it with other objects and operations.

attribute Serves to identify an object or operation. Required attributes for objects are *name*, *behavior*, and *domain*. Required attributes for operations are *name* and *algorithm*.

relation A complex attribute representing an association of an object or operation with other system objects and operations. Relations on objects include its *class* as well as sets of operations, *component objects*, *actor objects*, and *server objects*. Relations on operations include its object as well as sets of *modified objects*, *argument objects actor operations* and *server operations*.

class A complete design of an object which may be used as a template from which another object derives its characteristic structure and function.

name A string serving to identify an object or operation which must be unique within a context.

behavior A text description of an object's function when provided with certain stimuli.

domain A text description of the set of states to which an object may change.

actors A relation which denotes which objects or operations require services of some other object and operation pair.

servers A relation which denotes which objects or operations provide services to some other object and operation pair.

components A relation which denotes the parent/child relationships between objects.

arguments A relation which denotes which objects are required as arguments in the interface of an operation. This relation has the attribute *mode* which may be either input or output.

modifies A relation which denotes which objects are modified by the execution of an operation.

The model presented retains the function of the theoretical model, and adds the practical aspects of the programming model. The implementation of an object is not specified, nor is the syntax of the communication between objects limited to a specific method. Yet provisions are made for describing the interface between objects and operations of other objects, as well as for representing the fully recursive nature of real world objects.

2.2 Representing The Model.

Statically, an object-oriented design consists of a representation of a system in terms of the model previously described. As such, the object model could be easily represented in a relational database. However, a static representation is insufficient to fully communicate a complex behavior or the interrelationship between objects without a correspondingly complex textual narrative.

As an alternative to text, software developers have produced a plethora of graphical methods of representing software systems. A number of techniques have been proposed to represent an object-oriented design, some entirely new, some variations on more familiar methods.

Examples of graphical OOD methods were reviewed in an attempt to determine which kinds of representations most clearly represent the object-oriented model. Each method has its own strengths and weaknesses and represent one or more of the three basic views of a software design. These views include block diagrams, detail diagrams, and state transition diagrams.

Examples of block diagrams include the Booch diagram [9] [19] which identifies the objects and operations in the visible interface, and the dependencies between objects, but does not reveal which objects invoke which operations. The object diagram of Goddard Space Flight Center's General Object-Oriented Development methodology [42] is an even simpler example and appears to be a variation on structure charts [35]. These diagrams add the capability to show a clean parent-child or a virtual machine hierarchy of design objects.

Detail diagrams are typified by Modular design charts [53] and Buhr diagrams [11]. The former shows attribute types and operations within an object, as well as which components are used by specific object bodies. The latter link operations together directly through "control sockets" giving the flavor of a hardware wiring diagram.

The Interactive Ada Workstation (IAW) [22] implements Buhr diagrams and adds a petri net diagram for describing state transitions between operational objects. The AdaGraph[2] tool [14] which implements Cherry's PAMELA[3] methodology uses a process graph and adds a hierarchical subprogram graph. APEX, a system in development at the Air Force Wright Aeronautical Laboratories, also adds a petri-net diagram to its block diagram and process connection graph [2].

The SHARP methodology [12] uses a variety of pictographs representing all three views. Different diagrams are used for main program abstraction, object implementations, object interactions, object invocation, task rendezvous, subprogram data flow, data structures, and program unit operations.

All the methodologies referenced were developed specifically for designing Ada programs[4], resulting in many Ada unique distinctions. The graphical representation presented in this paper takes a more generic approach.

Rather than favor one view of design over another a multi-view approach is suggested consisting of three parts: a *block diagram*, an *interface diagram*, and a *control flow or state diagram*. Figure 2 shows an example of a simple design including these three views.

[2]AdaGraph is a trademark of The Analytic Sciences Corporation.

[3]PAMELA is a trademark of George W. Cherry

[4]The modular design charts were developed with both Ada and Modula2 in mind.

The block diagram used is similar to the high level object diagram of [42]. It depicts the objects in the system (at a particular level of detail) and the dependency relationships between them. Module dependency is shown by directed arrows to the servant or component objects in the graph. In the case of an actor/server relationship, messages or operation calls flow across the directed arrows.

The detail diagram is a modification of the modular design chart [53]. The requirements for depicting a "software bus" and separate component bodies are left out. In lieu of the implementation-oriented terms "package", "proc", "fn", and "type", objects begin with a capital letter, and operations begin with lower case.

A petri-net graph similar to the one found in APEX [2] is used to depict a state diagram or object interaction in the case of concurrent communicating objects.

The main purpose of graphics is to communicate the design *more clearly than does the text.* While the use of graphics is strongly advocated ("a picture is worth 1024 words"), a methodology so rigid that the graphic techniques drive the design, rather than good software engineering principles, can be counter productive. Thus the graphic representations offered should be implemented informally, rather than with such rigor that documentation costs exceed their expected benefit.

3 Overview of the Methodology

Webster defines a methodology as "a body of *methods*, *rules*, and *postulates* employed by a discipline: a particular procedure or set of procedures" [51]. In the previous sections postulates were offered regarding the decision-oriented nature of design, applicability of the object-oriented paradigm, and an object model for software design. This section describes the methods or steps to deriving a design using the object model. The methodology is based on providing rules or postulates (design heuristics) to support object-oriented design *decision making.*

The specific steps in the methodology were developed by first identifying the decisions involved in OOD from the literature and from experienced software engineers at AFIT. Thus the OOD *process* is not redefined, rather

it is presented in terms of *decisions* rather than the usual set of *products* associated with the design specification.

The OOD process is pictured in the concept map in Figure 3. The decision steps highlighted in the figure are defined as follows:

1. **Analyze** the problem and requirements specification to decide on an initial scope and a strategy for its solution.

2. **Identify** the abstract objects, operations, and their attributes from the solution strategy and requirements specification; then decide which are central to the solution strategy.

3. **Encapsulate** the objects, operations, and attributes into modules and determine the relationships, or interfaces, between those modules. In other words, decide which operations naturally go with which objects.

4. **Decompose** complex modules by repeating the process with objects or operations as separate problems, or begin detail design. Detail design requires deciding whether to construct modules from known components such as other objects, library modules, predefined functions or data types; or to produce an algorithmic description using psuedocode or flow diagrams.

OOD is unique in respect to what needs to be identified in analyzing the problem, how data structures and algorithms are encapsulated into system modules, and in how system modules are constructed from known, more general data types or classes. However, it should be clear that the main thrust of the decisions discussed here are basic to software design—regardless of the paradigm involved.

The following sections provide a more detailed descriptions to the decision steps of the methodology.

3.1 Analyze the Problem to Determine a Solution Strategy

3.1.1 Discussion.

The first decision the designer must make is in limiting the scope of the problem to be solved. In this step the initial context or scope is determined

for the subsequent steps. Abbott [1] and others [19] state that the problem must be reduced to a single sentence. A problem too complex to state in a single sentence probably requires a higher level of abstraction.

The problem statement should be determined from the problem space and stated in user-oriented terminology. It is important for the designer to interact with the user whenever possible in accomplishing this step. Using the *concept map* to elicit such problem-oriented information encourages this interaction and may communicate more effectively and ensure mutual understanding.

Concept maps should be developed from both the users and the requirements specification. The various results may then be compared and refined to provide a better understanding and statement of the problem. Working with the concept map of the problem, a map for a solution may be developed. The concept map may prove to be a better means of presenting the solution strategy than the single paragraph of [1] and [9], in the same way that the graphical structured specification [15] has proved more effective at communicating high level abstract requirements than a verbose textual document.

3.1.2 Summary of the Analysis Step.

1. Interview one or more users and develop concept maps of the problem.
2. Develop additional concept maps from the portions of the requirements specification which describe the system's functional requirements and entities at the desired level of abstraction.
3. Synthesize from the concept maps a single sentence statement of the problem.
4. Develop a single concept map which depicts a strategy for solving the problem.

3.2 Identify the Objects, Attributes, and Operations

3.2.1 Discussion.

Dave Bullman [27] states that finding the right objects is hard. He goes on to say that associating operations with the right objects is even harder. The

implied requirement of intuition and choice here indicates this as the next decision process.

A number of "rules of thumb" or heuristics have been suggested for both the identification of objects and encapsulation of objects with their attributes and operations. Thus this step consists of the application of such heuristics to identify and define the objects, attributes, and operations which apply within the scope and level of abstraction we are dealing with. Some valuable heuristics include the following:

Object Selection Criteria lists general software engineering heuristics such as information hiding, abstraction and inheritance for determining *good* objects [37].

Grammatical Analysis makes selections based on nouns and verbs [1].

Abstraction Analysis makes selections based on data flow diagrams [42].

Class Abstraction makes selections based on classes of physical objects [31].

Concept Analysis makes selections based on concept map entities and has the following steps:

1. Generate a first cut list of objects from the entities on the concept map. This is possible since the concept map is developed by a designer with OOD in mind.
2. Identify from the list of objects which are long-lived and which are transient. Transient objects tend to be operation arguments or local variables. Long-lived objects tend to represent abstract state machines.
3. Identify which objects are subordinate, natural components of, or clearly attributes of other objects and note these characteristics in the object's description. such in the object description.
4. Identify the action words in the relationships between entities as candidate operations. Describe the behavior of these actions as to what objects are modified, what information is required, which objects invoke the operations, and what other operations might they naturally require of other objects.

The primary objective of this step is *identification* along with some basic definition. Associating objects and operations is reserved until the encapsulation step. The elements in this step should come initially from the solution strategy unless the heuristics used require otherwise. It is difficult to initially scope a problem such that the lists of objects, and operations are complete, accurate, and without some spurious low level objects or operations having been defined. the analysis and identification steps may be repeated one or more times to arrive at a realistic scope of the problem and a complete set of objects and operations.

As identifiers of objects and operations, attributes should be associated with appropriate entities after they are identified. Listing object and operation attributes serves to define those entities in greater detail. The requirements document will often need to be consulted to fully describe program entities.

3.2.2 Summary of the Identification Step.

1. Apply one or more identification heuristics to identify the *set of objects* in the system at the scoped level of abstraction.

2. Analyze each object and describe its attributes and structure in the solution strategy. Check the requirements document for completeness and eliminate redundancy in the object list.

3. Apply one or more identification heuristics to identify the *set of operations* performed within the system at the scoped level of abstraction.

4. Analyze each operation to determine and generally define its stimulus/response attributes.

3.3 Encapsulate Objects, Attributes, and Operations into Modules

3.3.1 Discussion.

Deciding which operations should be associated with which objects is not as straight-forward as it may seem. Objects seldom behave independently of other objects. Consequently, observed behaviors may represent a complex

interrelationship among objects. A good example is the one hwere a drill is drilling a hole in a piece of metal [10], whether the operation *drill_hole* is an activity of the *drill_press*, *drill_bit*, or *sheet_metal* depends on the abstraction of those objects in the problem solution. Thus guidelines, rules, or heuristics are needed to guide the encapsulation of objects and operations in such a way as to produce *good* modules.

In choosing which objects and operations to encapsulate into modules, the interrelationships between modules are revealed. Those relationships or interfaces are specified by first determining the dependency between modules. A dependency exists whenever an operation of an actor or agent type object affects or requires an action by some other object. Rather than depict the dependencies only, the specific operations of an object required by each operation of each external object need to be diagramed. This includes identifying the attributes or arguments an operation requires to accomplish its function; and which attributes or internal objects are affected through such an operation under the stated conditions.

Heuristics for encapsulation include the following:

Modularity Rules define quality assessment of modules such as coupling and cohesion [37].

Object Classification requires identifying an object's operation as one of several general types such as actors or agents [10] [9].

Application Classification requires identifying an object or operation as one of a set of predefined types specified as a set common to the program application area [2].

Structural Classification requires identifying an object's structure as one of four general types (e.g. an abstract state machine) [9].

3.3.2 Summary of the Encapsulation Step.

1. Apply one or more encapsulation heuristics to the lists of objects and operations to determine a set of system modules.

2. Determine the interrelationships between modules and diagram the module dependencies.

3. Analyze each module dependency to determine and diagram the detailed interfaces between each dependent module's operations and the executors of those operations.

4. Refine the descriptions of the operations of each object in view of the various conditions under which it might be required of some other object and develop a state transition diagram if appropriate.

3.4 Decompose the Modules or Begin Detail Design

3.4.1 Discussion.

Decomposition deals with the question of how to construct each module. Should it be further decomposed, constructed from known components, or algorithmically defined via psuedocode or flow diagrams. This is the step in which inheritance may be applied since, at this point, a full description of each object at a given level of detail is available. To apply inheritance any earlier might result in shaping the solution to a set of preconceived notions rather than really solving the user's problem.

Inheritance is applied based on the object or module classifications made in the previous step. Such classifications are helpful, not only in determining module structure and behavior, but in identifying objects as instances of classes in the system, or as matching preexisting templates maintained in a class library. The decision to use inheritance is always a tradeoff between the cost of new development and the cost of modifications to existing templates.

Should inheritance fail to provide a solution to the design of a particular module, the module must be decomposed into smaller modules, or described at its lowest level as data structures and algorithms. Algorithmic description follows traditional methods useing Structured-English psuedocode or flow diagrams. Data structures which are operated on as a whole may be further described in a data dictionary.

All or part of a module may be decomposed. A module containing sets of objects and a set of operations, may have elements of those sets at their lowest level, and other elements of sufficient complexity to warrant decomposition.

Decomposition may take a variety of forms depending on the problem. For a functionally cohesive operation on a single object, conventional functional decomposition may be adequate. If aspects of the operation exhibit

concurrency, a process-oriented approach would be appropriate, with each sub-operation representing a single concurrent operation. Should the existence of other independent objects become apparent, an object-oriented approach might be better. In other words, the problem should lead to an appropriate design technique, rather than squeezing the problem into an unnatural methodology.

3.4.2 Summary of the Decomposition Step

1. Analyze the modules in the system for signs of common classes. If such a class hierarchy is apparent, indicate objects as instances of the class and further design the class.

2. Analyze the classification of modules in regard to existing generic structures or functions. Determine unique characteristics of such modules to determine cost effectiveness of redesign versus reuse.

3. Analyze the complexity of remaining modules and determine which module components must be further decomposed.

4. For each component which must be decomposed, determine the appropriate design method and proceed with the design. Appropriate flow diagrams, petri nets, structure charts etc. should be used to describe the design of components not accomplished in an object oriented fashion. Those components which require an object-oriented design, should be treated as new problems and designed using this methodology in an iterative fashion.

5. For each operation which need not be decomposed, describe its operation algorithmically using appropriate psuedocode or flow diagrams.

6. For each object or attribute which need not be decomposed, describe the data structure it represents.

4 Developing a Decision Aid for OOD

This section provides an overview of the steps taken for determining requirements and a top level design for a decision aid to implement the OOD

methodology. A brief discussion of a Smalltalk implementation for experimental purposes is also given.

In the field of decision support, requirements determination requires four steps: understanding the problem, selecting a kernel system to implement, developing a representation or model of the system in the form of storyboards, and describing the database and modelbase requirements to support the system. The storyboards and associated feature chart then serve as a top level design of the dialogue, database, and modelbase components of the decision aid.

4.1 Understanding the Problem

The problem is to provide a methodology for object-oriented design which addresses the decisions a designer must make. A solution was determined from concept maps of the OOD process and the resulting model and methodology were proposed.

4.2 Selecting the Kernel

The concept map of Figure 3 was used to show the OOD decision processes and to derive the feature chart shown in Figure 4. The feature chart depicts the support and interaction required by the four steps in the methodology. Storyboards were developed representing decisions and support requirements. The feature chart also shows supporting windows representing individual features provided by the storyboards.

The storyboards are linked together through the *main menu* which is be available from each storyboard for switching to any other storyboard. The main menu also provides a means of exiting the system and allows access to context sensitive help and the hook book. Several functions overlap. For instance, the object and operation definitions created in the Identification storyboard are used again in both subsequent storyboards.

The feature chart presents an overview of the features required by the kernel system. Consequently, only the high level, or external functions are shown. The storyboards and their descriptions reveal detailed requirements.

4.3 Representing the Kernel

Figures 6–10 in the appendix to this paper show the storyboards developed in the design of the dss kernel. In general, each storyboard contains at least three sub-windows or panes: a *features* pane, an *objects* pane, and a *text* pane. Selecting an element in the features pane causes a list of files or objects to appear in the objects pane. Selecting an element in the objects pane causes initialization of the text pane, or bring up a sub-window—either one of which the user will use to carry out some sub-step in the methodology.

4.4 Supporting the Kernel

4.4.1 The Database Requirements.

The database involves the storage, representation, and manipulation of design objects as well as on-line access to a requirements specification. The functions described in the storyboards require the ability to display graphics, text, and data dictionary information.

4.4.2 The Modelbase Requirements.

"Models are active relations and associations that govern decisions and actions in an organization" [28]. For the purposes of this paper, the object model of Section 2 and the heuristics and methodologies listed in Section 3 comprise the "relations and associations" which govern the design decisions in the OOD process. The system must be able to manage this information and present it to the user in a meaningful and timely manner.

4.5 A Prototype Decision Aid

Case studies of DSS usage show that "Key factors explaining successful development are a flexible design and architecture that permit fast modification and a phased approach to implementation" [44]. Thus although the methodology and initial requirements and design for a dss have been stated somewhat formally, only time and experience will tell whether or not the ensuing system will be accepted and of value to its users.

The suggested evolutionary design approach was applied to developing a prototype which would allow user response and feedback to determine the potential of these concepts.

It began with implementing the storyboards using the Smalltalk/V Object-Oriented Programming System. A single standard windowing style was used and as much functionality as possible was implemented such that even this first kernel system can be considered "a small but usable system to assist the decision maker" [44].

Implementation of the object model in Smalltalk consisted of declaring several new classes and selecting the data structures to represent the model. A simple relational approach was taken, directly implementing the relations implied by Figure 11 derived from the model description.

The primary data structure was implemented as a dictionary of relations with the name of the relation as the key to the dictionary. Each relation was then implemented as an ordered collection of dictionaries with two or more associations of pointers to objects or operations. Figure 12 graphically depicts this structure.

Standard data structure operations were implemented to hide this structure from the using storyboard features. Additional special operations were then added to support unique database accessing requirements to simplify code in the storyboard operations.

The modelbase was implemented as a context sensitive set of text help files representing design heuristics and methodology instructions. The executive control module maintains lists of help files which can be edited, removed, or added to by the users. Each storyboard contains its own list of heuristic files developed from the examples discussed previously.

To aid in evolutionary development the Hook Book was fully implemented as a separate object with its own browser for entering, adding, and removing entries.

5 Conclusions

This paper has only briefly introduced an adaptive approach at developing software support tools and environments. While the specific target was an object-oriented design methodology, the concepts regarding adaptive, evolu-

tionary design apply to systems supporting many design methods and life cycle phases. The central hypothesis of this effort is that design is essentially a decision process and if *good* systems are to be produced, *good* decisions must be made. The software engineering community must take as hard a look at improving the engineers decision making capabilities as it does in representing those decisions with flashy graphics and powerful databases.

References

[1] Abbott, R. J. "Program Design by Informal English Descriptions," *Communications of the ACM*, 26, 11: 882–894 (November 1983).

[2] Air Force Wright Aeronautical Laboratories. *APEX Users' Guide.* AFWAL, Wright-Patterson AFB, CO., 1987.

[3] Alabiso, 200 Bruno. "Transformation of Data Flow Analysis Models to Object- Oriented Design," *OOPSLA '88 Conference Proceedings, ACM SIGPLAN Notices*, 23, 12: 335–353 (September 1988).

[4] Alford, Mack. "SREM at the Age of Eight; the Distributed Computing Design System," *IEEE Computer*, 18, 4: 36–46 (April 1985).

[5] Andriole, Stephen J. and others. *Storyboarding for C2 Systems Design: A Combat Support System Case Study.* Unpublished paper, George Mason University & International Information Systems, Inc. 802 Woodward Road, Marshall, VA 22115, undated.

[6] Balzer, R. and others. "Software Technology in the 1990s: A New Paradigm," *IEEE Computer*, 16, 11: 39–45 (November 1983).

[7] Bohm, C. and Jocopini, G. "Flow Diagrams, Turing Machines, and Languages with only Two Formal Rules," *Communications of the ACM*, 9, 5: 336–371 (May 1966).

[8] Booch, Grady. *Software Components with Ada.* Menlo Park: The Benjamin/Cummings Publishing Company, Inc., 1987.

[9] - - - - -. *Software Engineering with Ada*(Second Edition). Menlo Park: The Benjamin/Cummings Publishing Company, Inc.,1986.

[10] Bralick, William A. Jr. *An Examination of the Theoretical Foundations of the Object-Oriented Paradigm.* MS Thesis, AFIT/GCS/MA/88M-01, School of Engineering, Air Force Institute of Technology (AU), Wright Patterson AFB OH, March 1988.

[11] Buhr, R. J. A. *System Design with Ada.* Englewood Cliffs: Prentice-Hall Inc., 1984.

[12] Byrne, William E. and others. *Structured Hierarchical Ada Representation Using Pictographs (SHARP) Definition, Application, and Automation.* Technical Report Prepared For Electronic Systems Command, Deputy for Development Plans, Hanscom AFB, Massachusetts. Cambridge: Arthur D. Little, Inc. Program Systems Management Co., September 1986.

[13] Cox, B. *Object-Oriented Programming: An Evolutionary Approach.* Reading: Addison-Wesley, 1986.

[14] Crawford, Bard S. and Jazwinski, Andrew H. "The AdaGRAPH^{TM} Tool for Enhanced Ada Productivity," *IEEE Transactions on Software Engineering*, SE-12, 5: 664–670 (May 1986).

[15] Demarco, Tom. *Structured Analysis and System Specification.* Englewood Cliffs: Prentice-Hall Inc., 1978.

[16] Diedrech, Jim and Milton, Jack. "An Object-Oriented Design System Shell," *OOPSLA '87 Conference Proceedings, ACM SIGPLAN Notices*, 22, 12: 61–67 (December 1987).

[17] Digitalk Inc. *Smalltalk/V Tutorial and Programming Handbook.* Los Angeles: Digitalk Inc., 1986.

[18] Department of Defense. *Requirements for the Programming Environment for the Common High Order Language (STONEMAN).* Washington: Government Printing Office, 1980.

[19] EVB Software Engineering,Inc. *An Object Oriented Design Handbook for Ada Software.* Fredrick: EVB Software Engineering, Inc., 1986.

[20] Ewing, Juanita J. and Wirfs-Brock, Rebeccca. "Smalltalk isn't Meaningless Chatter," *Computer Design*, 26, 1: 76–79 (January 1987).

[21] Freedman, Roy S. "The Common Sense of Object-Oriented Languages," *Computer Design*, 22, 2: 111–118 (February 1983).

[22] General Electric Corporation Research and Development Division. *Users' Guide : Interactive Ada Workstation, Prototype Version 1.0.* DOD Contract No. F33615-85-C-1755, General Electric Co., August 1986.

[23] Hartrum, Thomas C. and Lamont, Gary B. "Development of a Comprehensive Software Engineering Environment," *Space Operations Automation and Robotics Conference*, Houston (September 1987).

[24] Jackson, Michael. *System Development*, Englewood Cliffs: Prentice Hall Inc., 1983.

[25] Keen, Peter G. W. "Adaptive Design for Decision Support Systems," *ACM/Database*, 12, 2: 15–25 (Fall 1980).

[26] Kelly, John C. "A Comparison of Four Design Methods for Real-Time Systems," *Proceedings of the 9th International Conference on Software Engineering.* 238–251. Washington: Computer Society Press of the IEEE, 1987.

[27] Kerth, Norman L. and others. "Summary of Discussions from OOPSLA-87's Methodologies & OOP Workshop," *Addendum to the Proceedings OOPSLA '87, ACM SIGPLAN Notices*, 23, 5: 9–16 (May 1987).

[28] Konsynski, Benn and Sprague, Ralph H. Jr. "Future Research Directions in Model Management," *Decision Support Systems*, 2: 103–109 (1986).

[29] Korth, Henry F. and Silberschatz, Abraham. *Database System Concepts.* New York: McGraw-Hill, Inc., 1986.

[30] Liang, Ting-peng. "User Interface Design for Decision Support Systems: A Self-Adaptive Approach," *Information & Management*, 12: 181–193 (December 1987).

[31] Lorensen, W. "Object-Oriented Design," *CRD Software Engineering Guidelines*, General Electric Co., 1986.

[32] Magel, Kenneth. "Principles for Software Environments," *ACM SIGSOFT Software Engineering Notes*, 9, 1: 33–35 (January 1984).

[33] Nassi, I. and Schneiderman B. "Flowchart Techniques for Structured Programming, " *SIGPLAN Notices ACM*, 8, 8: 12–26 (August 1983).

[34] Novak, Joseph D. and Gowin, D. Bob. *Learning How to Learn.* Cambridge: Cambridge University Press, 1984.

[35] Page-Jones, Meilir. *The Practical Guide to Structured Systems Design.* New York: Yourdon Press, 1980.

[36] Pascoe, Geoffrey A. "Elements of Object-Oriented Programming," *Byte*, 11, 8: 139–144 (August 1986).

[37] Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (Second Edition). New York: McGraw-Hill Book Company, 1987.

[38] Riedel, Sharon L. and Pitz, Gordon F. "Utilization-Oriented Evaluation of Decision Support Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-16, 6: 980–006 (November 1986).

[39] Ross, Douglas T. "Applications and Extensions of SADT," *IEEE Computer*, 18, 4: 25–34 (April 1985).

[40] - - - - -. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, SE-3, 1: 16–34 (January 1977).

[41] Seagle, John P. and Belardo, Salvatore. "The Feature Chart: A Tool for Communicating the Analysis for a Decision Support System," *Information & Management*, 10, 1: 11–19 (January 1986).

[42] Seidewitz, Ed and Stark, Mike. "Towards a General Object-Oriented Software Development Methodology," *ACM Ada Letters*, 7, 4: 54–67 (August-September 1987).

[43] Simon, H. *The New Science of Management Decision.* New York: Harper & Row, 1960.

[44] Sprague, Ralph H. Jr. and Carlson, Eric D. *Building Effective Decision Support Systems.* Englewood Cliffs: Prentice-Hall, Inc., 1982.

[45] Stay, J. F. "HIPO and Integrated Program Design," *IBM System Journal*, 15, 2: 143–154 (1976).

[46] TRW Defense Systems Group. *Distributed Computing Design System (DCDS) Methodology Guide (Ada Version).* Huntsville: TRW System Development Division, October 1987.

[47] Valusek, John R. *The DSS Cube.* Class lecture in OPER 652, Decision Support Systems. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1987.

[48] - - - - -. *Concept Mapping.* Class handout distributed in OPER 652, Decision Support Systems. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1987.

[49] - - - - -. *The Hook Book.* Class lecture in OPER 652, Decision Support Systems. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, April 1987.

[50] Warnier, J.D. *Logical Construction of Systems.* New York: Academic Press, 1975.

[51] Webster. *Webster's New Collegiate Dictionary.* Springfield: G. & C. Mirriam Company, 1981.

[52] Wegner, Peter. "Dimensions of Object-Based Language Design," *OOPSLA '87 Conference Proceedings, ACM SIGPLAN Notices*, 22, 12: 168–182 (December 1987).

[53] Wiener, Richard and Sincover, Richard. *Software Engineering with Modula-2 and Ada*, New York: John Wiley & Sons, Inc., 1984.

[54] Wirth, N. "Program Development by Stepwise Refinement," *Communications of the ACM*, 14, 4: 221–227 (April 1971).

A Figures

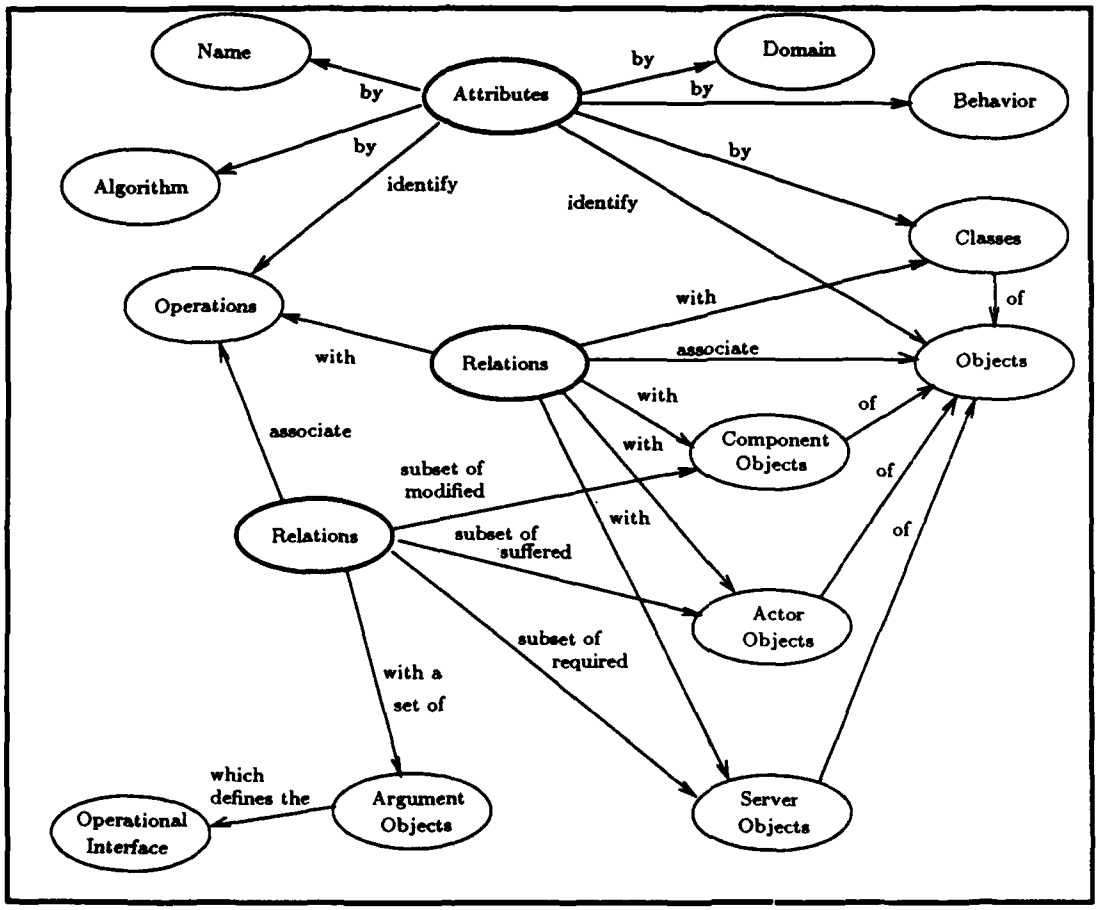


Figure 1. An Object Model for Design

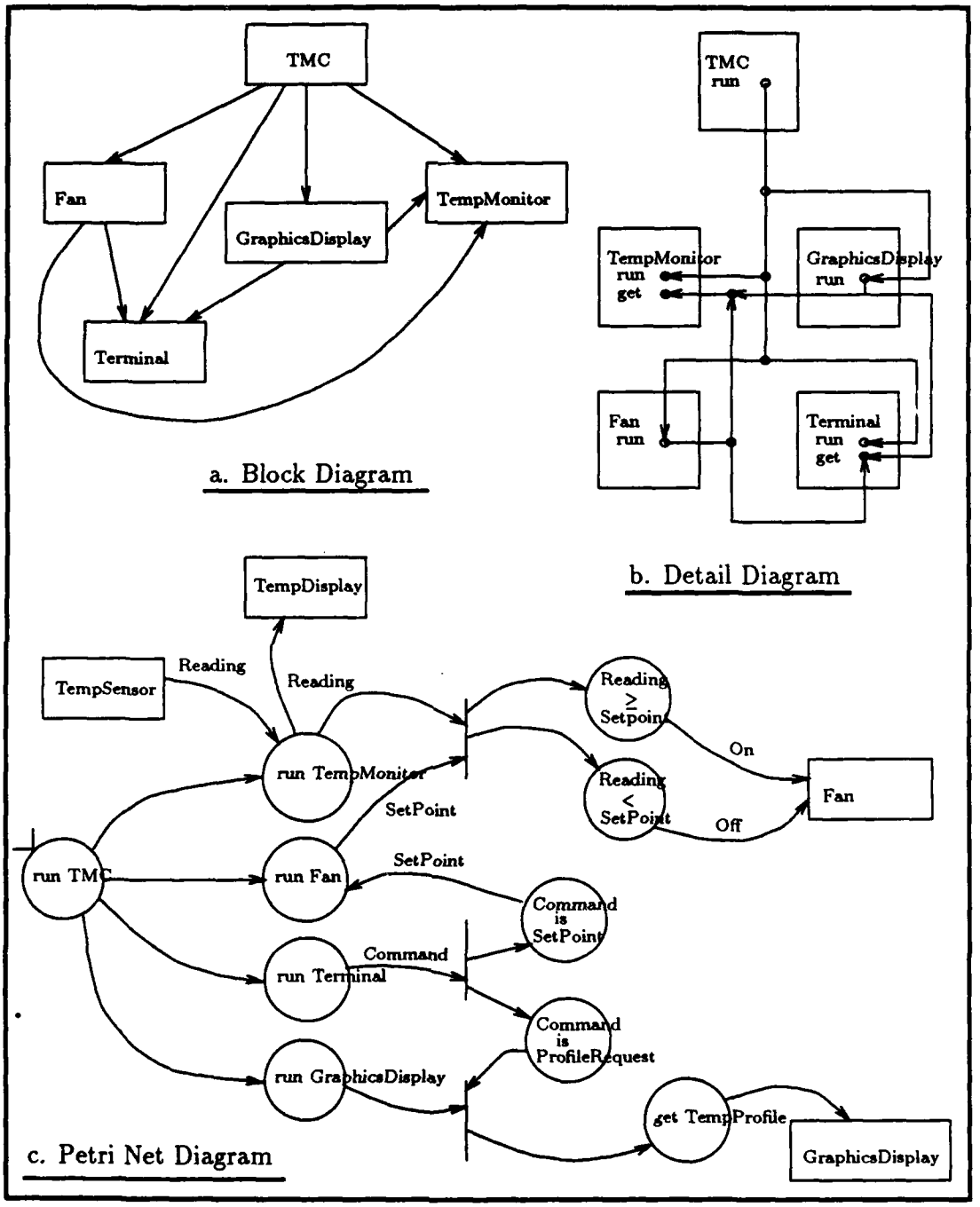


Figure 2. The Three Views of a Language Independent OOD

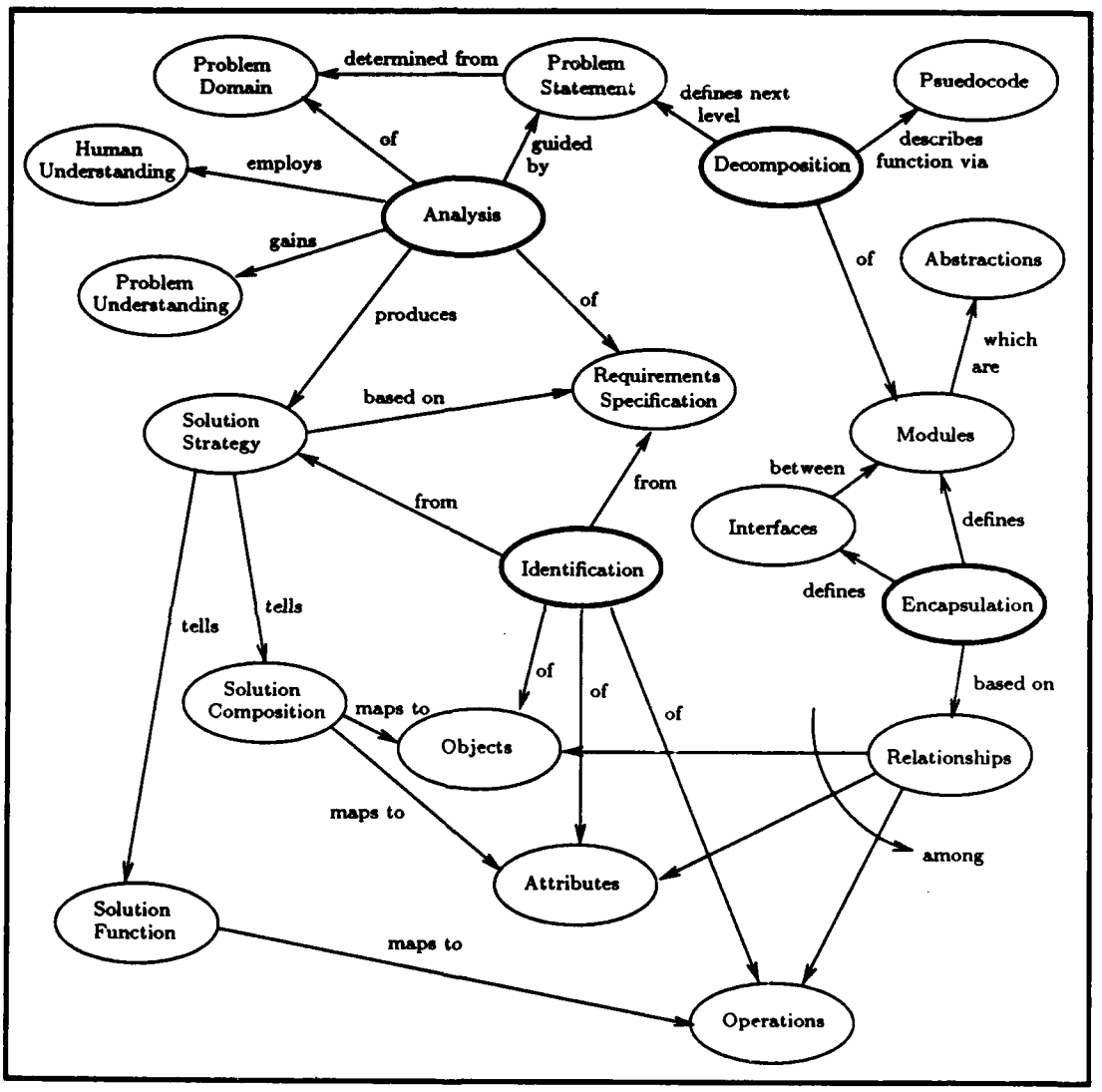


Figure 3. The Relationship Between Object-Oriented Design Decision Steps

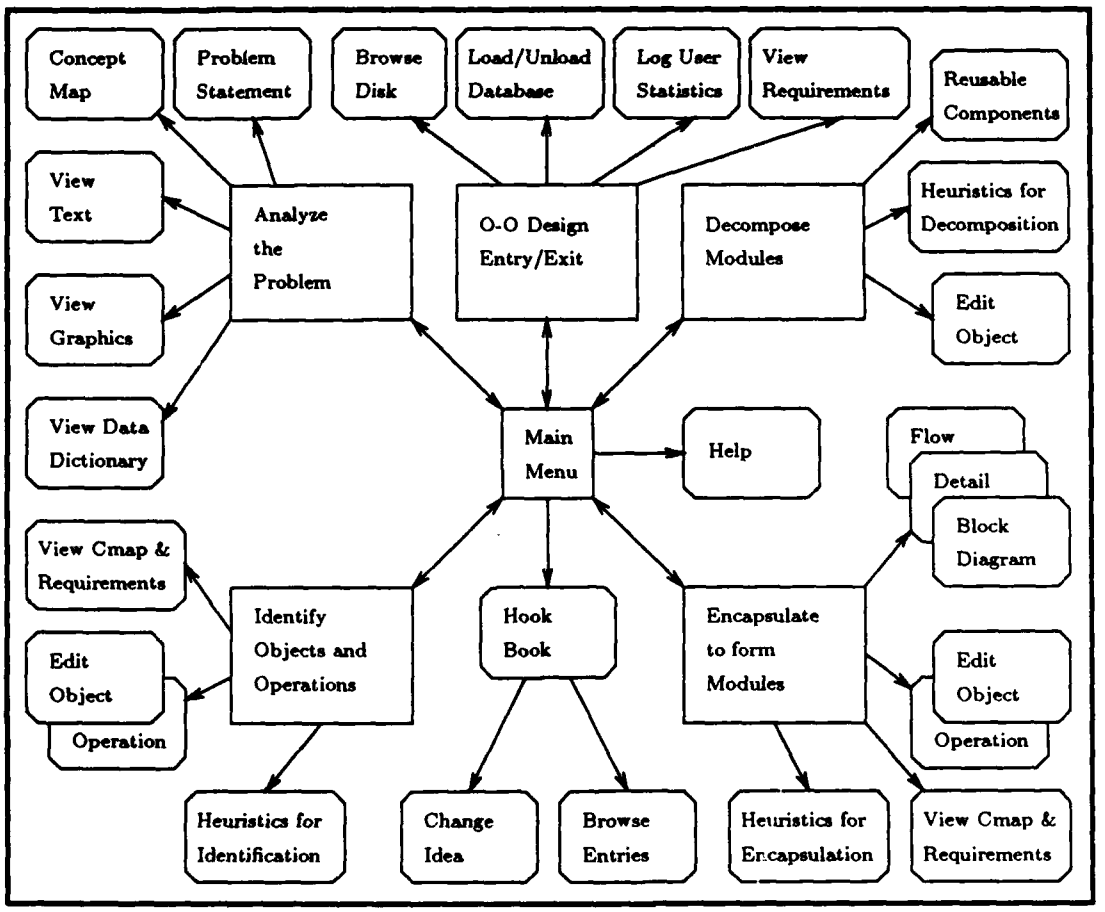
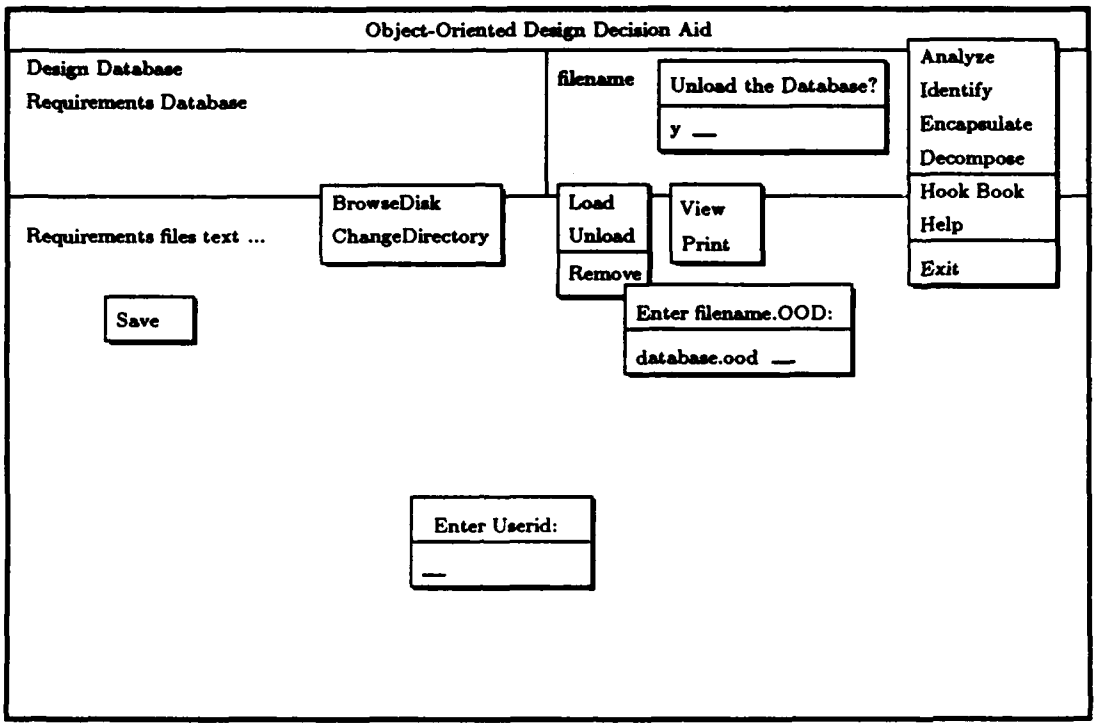


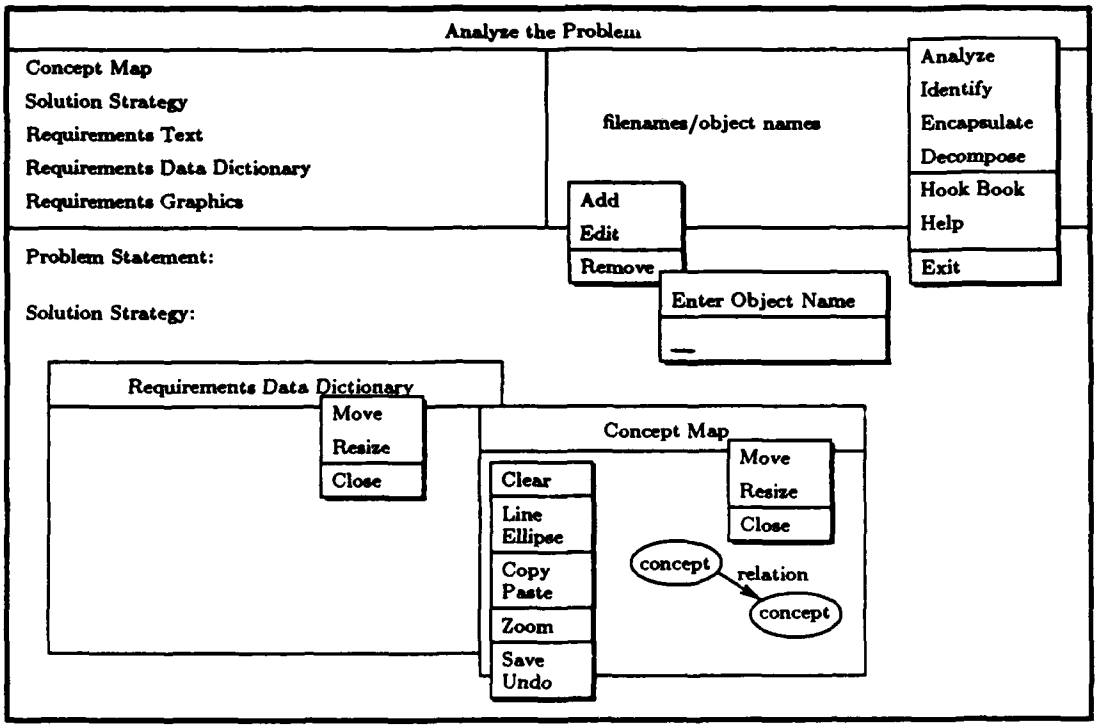
Figure 4. Feature Chart for the OOD Decision Aid



The Entry/Exit display is described as follows:

1. The user will initailly be prompted for a userid. Login/logout times will be automatically recorded.
2. The main menu will allow activating other storyboards, the Hook Book, context sensitive help, or exiting the system.
3. Selecting entries from the features pane produces the following results:
   a. Selecting the DesignDatabase causes database files to be listed in the objects pane.
   b. Selecting the Requirements Database causes all requirements files to be listed in the objects pane.
4. The features pane popup will allow the following:
   a. Activation of a disk browser facility.
   b. Prompting the user for a new default directory for the database, requrements, or help files.
5. The objects pane popup will allow loading and unloading the design database; or printing/viewing requirements files.
6. The text pane will provide the following:
   a. Initial instructions on startup. Editing and saving instructions.
   b. Viewing requirements files.

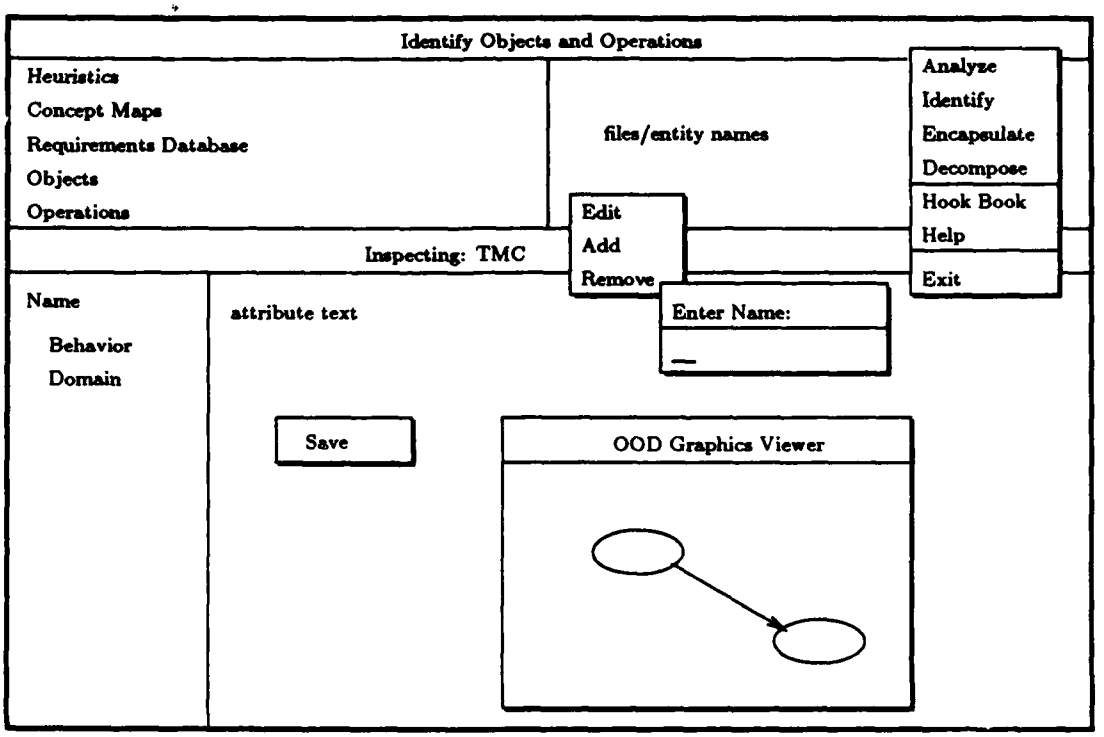
Figure 5. Storyboard: Entry/Exit for the OOD Decision Aid



The Analysis display is described as follows:

1. The main menu may be activated. Exit will return user to Entry/Exit.
2. The user will be prompted for the name of the object to be designed.
3. Selecting entries in the features pane results in the following actions:
   a. Selecting the ConceptMap entry causes concept map object names to be listed in the objects pane.
   b. Selecting Solution Strategy causes object names to be listed in the objects pane.
   c. Selecting RequirementsText, DataDictionary, or Graphics entries causes the corresponding files to be listed in a sub-window.
4. Selecting entries in the objects pane results in the following actions:
   a. Selecting a concept map will allow editing, saving or removing concept maps from objects.
   b. Edit will open a graphic drawing window for creating, editing, and saving concept maps.
   c. Selecting Edit will bring up a sub-window for editing the concept map.
   d. Selecting an object for a Solution Strategy will display the text to the text pane or format the text pane for creation.
   e. Selecting a requirements file will activate a sub-window for viewing requirements data.
5. The concept map sub-window will provide the capability to generate and edit graphics representations of concept maps.

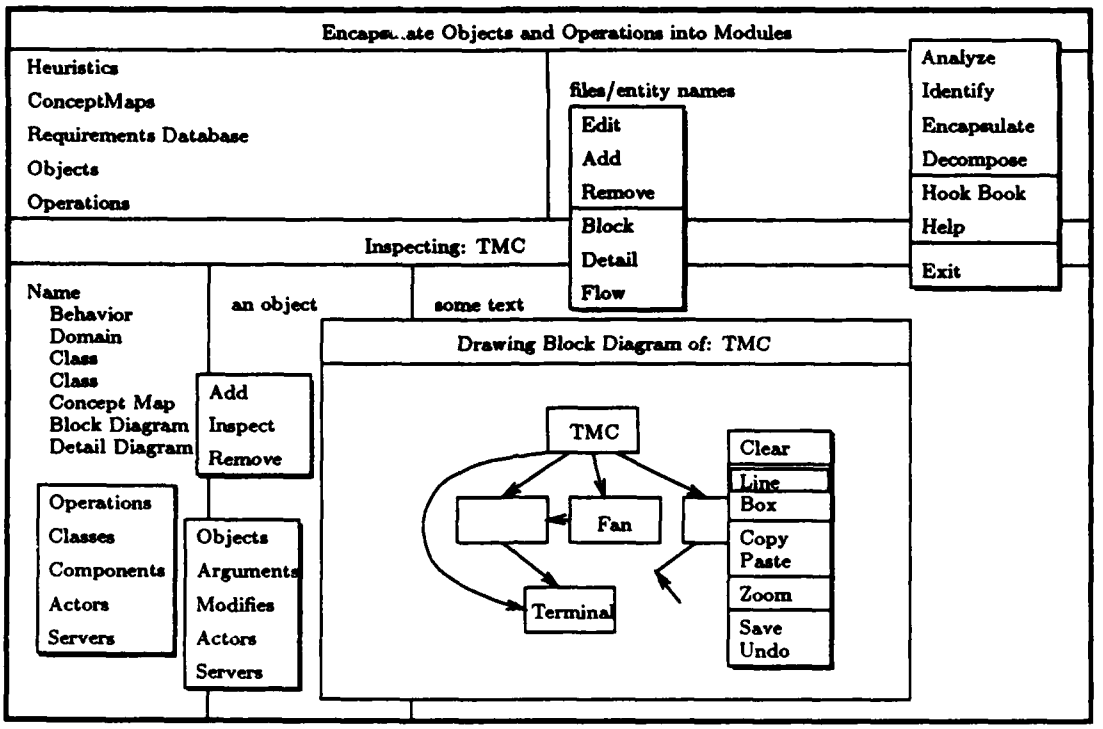
Figure 6. Storyboard: Analyze the Problem



The Identification display is described as follows:

1. The main menu may be activated. Exit will return user to Entry/Exit.
2. Selecting entries from the features pane produces the following results:
   a. Selecting Heuristics causes a Help window to open for viewing/editing heuristics.
   b. Selecting Requirements, or ConceptMap causes file or object names to be listed in the objects pane.
   c. Selecting Object or Operation causes database entries to be listed in the objects pane.
3. Selecting an entry in the objects pane produces the following results:
   a. Selecting a concept map, or a requirements source file, will activate the appropriate sub-window for viewing only.
   b. Selecting an object or operation name will activate a popup for Adding, Editing, or Removing entities from the database. operation.
   c. Selecting Add will open a database browser in the text window.
4. The Database Editor will provide the following capabilities:
   a. An attribute pane will provide the ability to add, inspect or remove attributes.
   b. A text pane will allow editing an entry's attributes.

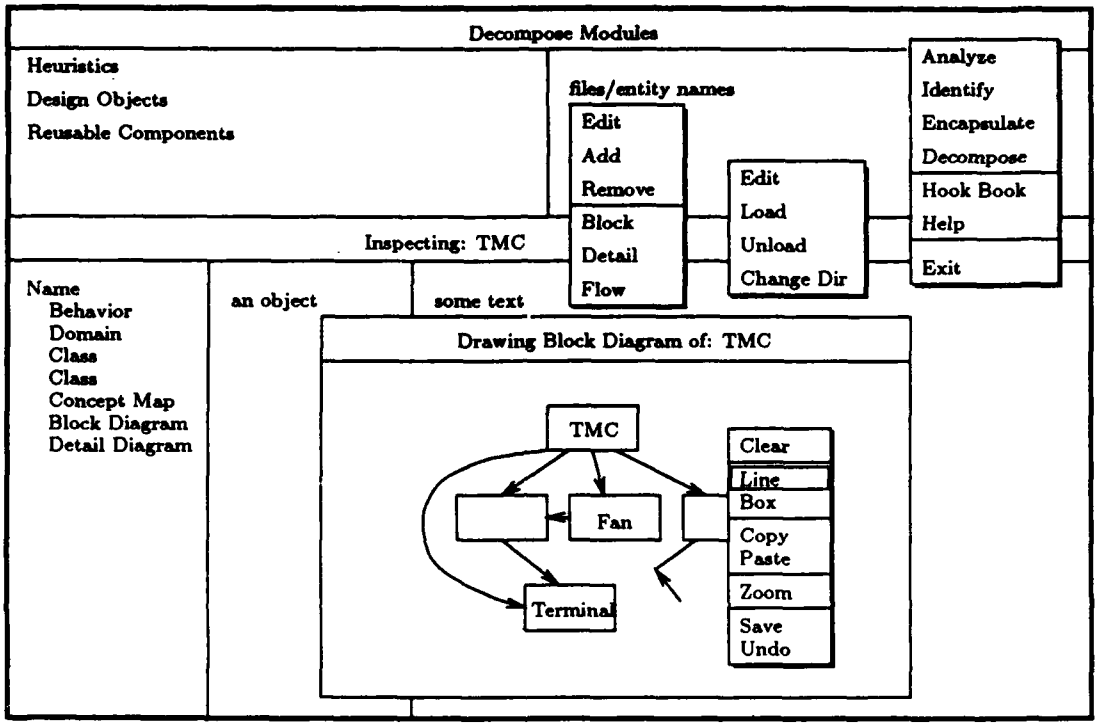
Figure 7. Storyboard: Identify the Objects and Operations



The Encapsulate display is described as follows:

1. The main menu may be activated. Exit will return user to Entry/Exit.
2. Selecting entries from the features pane results in the appropriate file or object names being listed in the features pane.
3. Selecting entries from the objects pane results in activation of the appropriate sub-window—except for objects and operations.
4. Selecting an object or operation from the objects pane opens a pop-up for selecting Editing the object/operation or creating Block, Detail, or Flow diagrams.
5. Selecting Edit opens a Database Browser with an additional list pane for forming relations.
6. The Database Browser provides the following additional capabilities:
   a. A context sensitive pop-up menu will list the possible relations for either an object or operation.
      Object: Operations, Components, Actors, Servers, Classes.
      Operation: Objects, Arguments, Modifies, Actors, Servers.
   b. Selecting a relation causes a second pop-up to appear for selecting Add, Remove, or Inspect.
   c. Selecting Add lists all appropriate objects or operations from wich to select in the list pane.
   d. Selecting inspect lists all defined objects or operations in the relation for the selected object. Selecting one opens an Inspector window on the object.
7. Selecting Block, Detail, or Flow results in activation of a graphics sub-window similar to the concept map sub-window.
8. Graphics sub-windows will provide for creation of rectangles or circles or other shapes as appropriate to the type of graphic being developed.

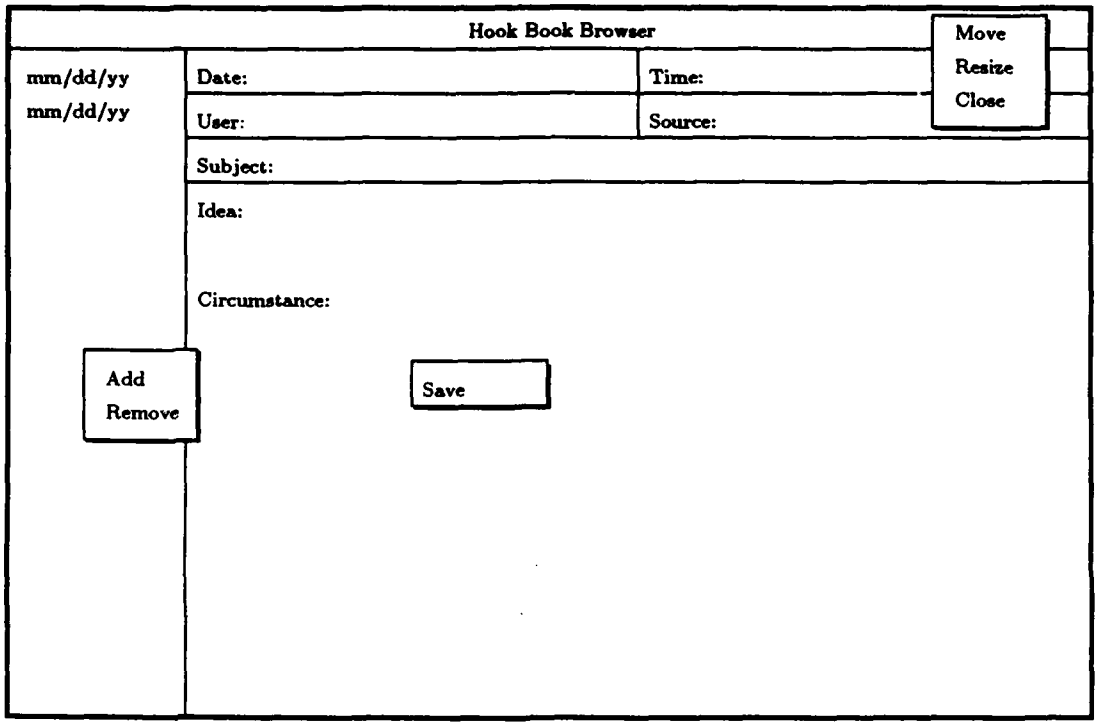
Figure 8. Storyboard: Encapsulate the Objects with their Operations



The Decomposition display is described as follows:

1. The main menu may be activated. Exit will return user to Entry/Exit.
2. Selecting entries from the features pane produces the following results
   a. Selecting Heuristics opens a help window for decomposition heurictics.
   b. Selecting DesignObjects lists objects in the files pane in a component hierarchy. Selecting objects opens a Database Browser as with the Encapsulation storyboard.
   c. Selecting ReusableComponents lists reusable components database files in the objects pane.
3. The objects pane provides a pop-up for load/unloading reusable components databases, and editing objects and their associated graphics.

Figure 9. Storyboard: Decompose the Modules



The Hook Book Browser display is described as follows:

1. A popup menu will provide the ability to move, resize, or close the browser.
2. A list pane will list all hook book entries by date and time.
3. Selecting an entry will cause the corresponding mini-panes in the rest of the window to be updated from the hook book entry.
4. Selecting Enter will cause the Date, Time, Userid, and Storyboard called from to automatically be entered in the labeled min-panes.
5. The user will be immediately prompted for a subject.
6. The text pane will provide for entering and saving the idea and circumstances.

Figure 10. Storyboard: The Hook Book Browser

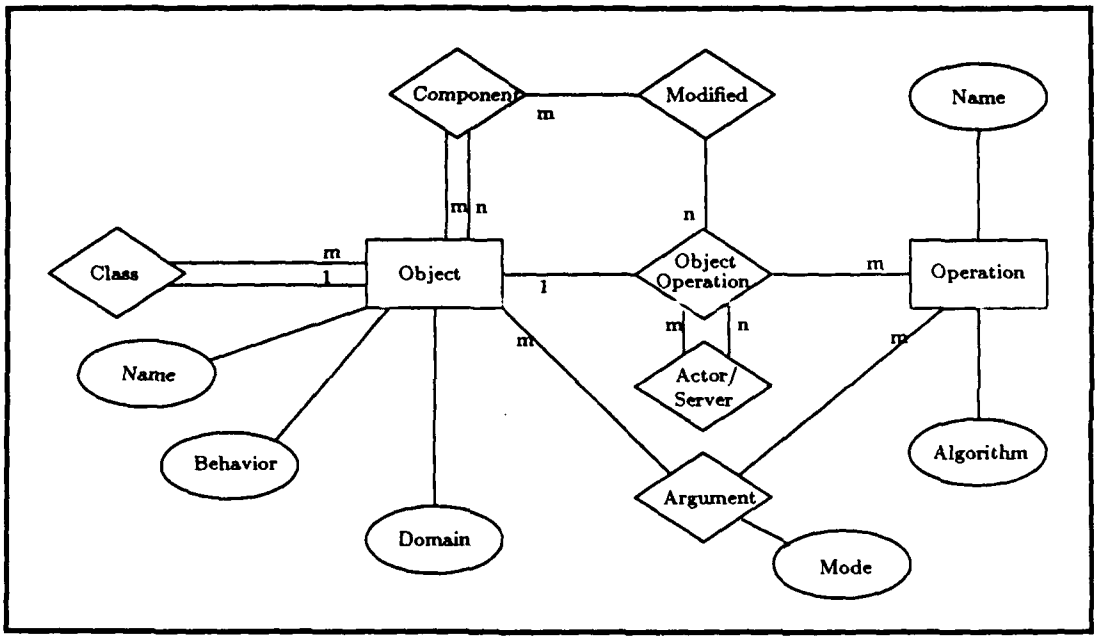


Figure 11. An E-R Diagram for the Object Model

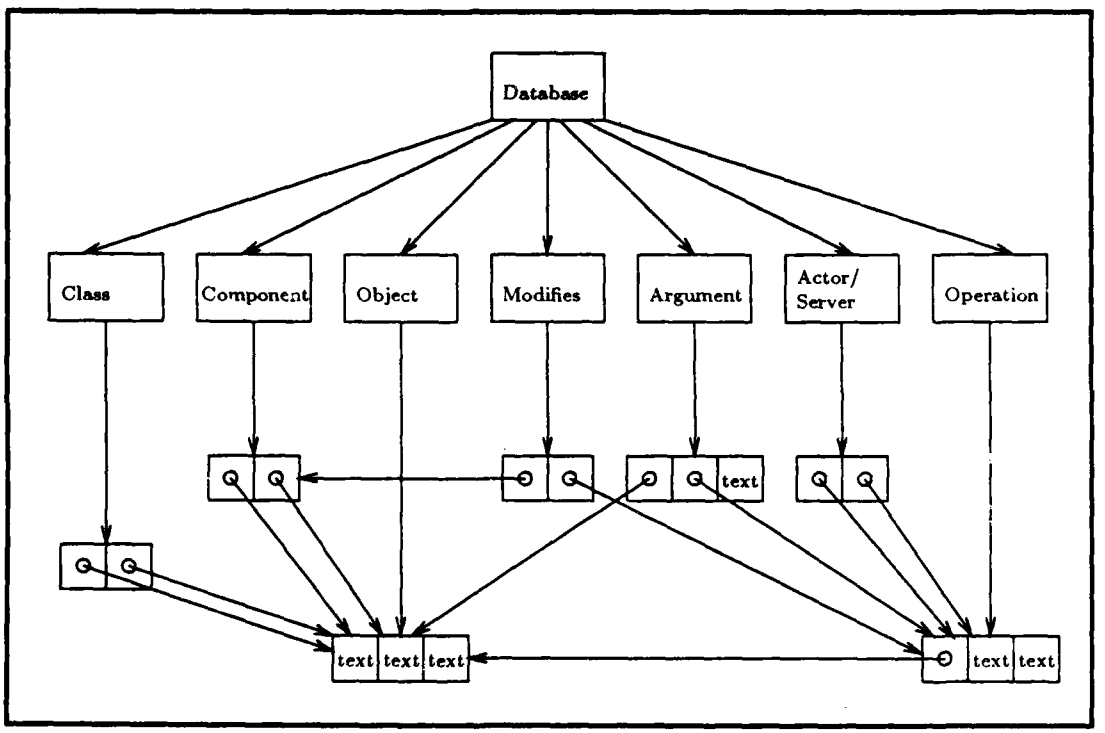


Figure 12. OOD Database Internal Structure

Appendix B. *OOD Decision Aid Programmer's Manual*

A Decision-Aid

for

Object-Oriented Design

A Decision Aid for Object-Oriented Design

PROGRAMMER'S MANUAL

Captain Patrick D. Barnes, USAF

December 16, 1988

1 Introduction

The OOD Decision Aid is a decision support system for developing an object-oriented design of computer software. The system is composed of a dialogue component, a requirements database, a design database, and a modelbase.

The system's dialogue component is written in the Smalltalk/V Object Oriented Programming System. The dialogue provides access to and manipulation of the database and modelbase and provides a graphic user interface. Tools are provided for browsing the database, hook book, and help information, as well as for developing graphic representations of the design.

The dialogue consists of a series of "storyboards" or screen displays representing decisions which must be made by the user in carrying out the design process. Each storyboard consists of a top pane with the storyboard label, a features pane listing the functions which may be performed, an objects pane listing the objects or files which may be manipulated by executing the selected feature, and a text pane which acts as a work area for the feature or for displaying useful information. Each pane has one or more menus which may be activated to control execution of particular functions.

The requirements database is a three view representation of a requirements specification developed using the Software Requirements Engineering Methodology (SREM) with the Distributed Computing Design System (DCDS). Text, data dictionary, and graphics representations are accessible by the dialogue.

The design database is a Smalltalk object which may be loaded, unloaded, and manipulated by the dialogue. It consists of a set of relations representing objects and operations and their attributes. Relations also represent various relationships between objects and other objects and operations.

The modelbase consists of text files representing software engineering heuristics and methodology guidance for the user. The dialogue reads, displays, and modifies these files and saves changes to disk. In addition, files may be created "off line" and made available to the system through the help and heuristics facilities.

2 Tool Set

The decision aid provides the following central and support tools:

Executive Control: The *OODDecisionAid* class is implemented as the controlling class of the decision aid. It provides controlled access to the storyboards as well as the hook book and help facilities. Each storyboard is implemented as a subclass of the abstract class *OODStoryboard* and controls its own sub-windows and features. The *OODDecisionAid* class executes the functions of the main menu (accessed from each storyboard's top pane) and the top menu of most sub-windows to eliminate redundancy and provide a single point of control over open windows.

User Log: The user is prompted for a userid at start- up and the start and stop times the user was on the system are automatically recorded. The User Log can be accessed off-line and can be saved or reloaded if the software is rehosted.

Hook Book: The user may record problems, suggestions, or comments during on-line operation through a hook book browser. The browser automatically logs the userid, time and date, current storyboard, and

prompts for a subject. The user can make entries, browse current entries, and delete outdated entries. Entries are listed in date/time order.

Help: Context sensitive help and heuristics files are provided via a browser which allows the user to select, add, or delete specific help information. File contents may be edited by the user and saved, thus making the system somewhat user tailorable.

Graphics: A drawing tool is provided for developing, viewing, storing, and retrieving bit-mapped graphic images of the design. To reduce the Smalltalk image size, all graphics are loaded and unloaded to disk files.

Requirements Browser: A browser is provided for retrieval and display of DCDS graphics, text, and data dictionary data. Methods are also provided to give the decision aid lists of files for display in the objects pane of a storyboard. Should a different system for requirements be desired, only this class need be modified or a subclass be developed for the new methodology.

Database Browser: A browser is provided for retrieval, display and manipulation of relations between database objects and operations. The browser can list a selected object's attributes, list related objects, and show the state of selected attributes. Menus provide the capability to create or delete relations or modify an attribute's state.

3 Configuration

3.1 Hardware

The system was developed using a Zenith Data Systems Z-248 microcomputer with a hard disk drive and a Microsoft compatible mouse. EGA graphics were available but are not required. The system is encumbered by the 640K DOS limit so 640K is recommended as a minimum. While an IBM/PC or XT compatible microcomputer should work, an 80286 based machine is recommended. If available, a two-megabyte RAM disk will improve

system performance. The Smalltalk/V user's manual describes how to run Smalltalk/V using a RAM disk.

3.2 Software

Smalltalk/V or Smalltalk/V286 may be used with the following applications loaded: FreeDrawing (provided with Smalltalk/V); Doscall, Loader, and Zoom (provided with Goodies 1). The Application Browser provided with Goodies 3 was used to control changes made for the application. However, it is not required for simply loading and executing the decision aid.

To capture and display DCDS graphics, the VTEK Textronix terminal emulator was used. The PLAY program and its support files are required to display the TKF files. As shown in the Files section, these files must be located in the Smallalk/V execution directory.

The OOD Decision Aid application is loaded from within Smalltalk/V in the same way as other Smalltalk applications. The disk browser is used to access the file *ooddcsna.prg* and the preface in the file contains instructions for loading the application.

4 Files

The Hard disk should be configured with a specific Smalltalk/V root directory and three sub-directories with the following files:

smaltalk

Smalltalk/V files
image
sources.sml
go
change.log
doscall.com
v.exe
v2ndpart.exe

VTEK files
play.exe
teksetup.dat
config.vtk
matrix.fnt

Global Support Objects
userlog.obj
hookbook.obj
helplist.obj

help

Help files
entryext.hlp
anlyssst.hlp
idntfcatn.hlp
encpsltn.hlp
dcmpstns.hlp
smalltlk.hlp
cncptmpp.hlp
oodmthdl.hlp

Heuristics files
abstrctn.hlp
addcmpfr.hlp
applctnc.hlp
clssabst.hlp
cncptanl.hlp
detldsgn.hlp
grmmtcla.hlp
inhertnc.hlp
mdlrtyrl.hlp
objctcls.hlp
objctslc.hlp
strctrl.hlp

analysis

Text files
tmc.txt
reqmnts.txt

Data Dictionary files
tmc_all.rdd
tmc_alph.rdd
tmc_data.rdd
reqmnts.rdd

Graphics files
tempnet.tkf
termnet1.tkf
termnet2.tkf
time_net.tkf
ctrlfan.tkf
crtpltfl.tkf

ood

Design Database
design.ood

Design Graphics Objects
tmc.map
tmc.blk
tmc.dtl
tmc.flw

5 Utilities

The following utilities were written to provide offline access to the system.

OODDecisionAid loadGlobals: Executing this statement will load the context sensitive help list, hook book entries, and user log from disk

files. The message **unloadGlobals** unloads those objects to the corresponding files named in the previous section. In both cases, files will be found/saved in the directory from which Smalltalk/V is loaded.

OODDecisionAid userLog: Executing this statement will return the user log for further manipulation such as printing out system usage.

OODStoryboard database: Executing this statement will return the current database object for off-line inspection and testing.

OODStoryboard clearDatabase: Executing this statement will set the database stored in the class variable of the OODStoryboard to nil for testing the database initialization code.

Appendix C. *DCDS Requirements Specification*

A Temperature Monitor Controller

C.1 Preliminary System Specification

C.1.1 Description This specification describes the requirement for a simple temperature controller. The computer system is connected to a temperature sensor from which it receives temperature readings. These readings must be displayed on a connected digital readout device. An ON/OFF signal is required to control an attached fan. An attached graphics screen allows temperature profiles to be displayed upon command. A terminal interface allows the user to input a setpoint value or to request a temperature profile display for a specified time period. The fan will be turned on whenever the temperature is above the setpoint, and off when the temperature is below the setpoint. The scope of this effort is the development of the software to support the specified hardware.

C.1.2 System Interface

C.1.2.1 Temperature Sensor. This system will receive temperature in a digital form from an attached temperature sensor subsystem.

1. *Sensor Trigger* A temperature report will be sent by the temperature sensor subsystem whenever it receives a temperature request.
2. *Physical Interface* The temperature sensor subsystem is connected via a 9600 baud full duplex RS-232 connection.
3. *Request Format* A temperature request to the temperature sensor subsystem consists of the ASCII sequence ESC]R$ from the computer.
4. *Error Handling* Any characters received by the temperature sensor subsystem *not* in the form ESC]R$ will be ignored.
5. *Temperature Report Format* A temperature report from the temperature sensor consists of the ASCII sequence

   ESC]Txxx.yyy$

   where xxx.yyy is a seven digit string comprising the current temperature expressed in degree C.
6. *Response Time* The temperature report will be sent (first byte transmitted) within 1.0 seconds of receipt of a temperature request (last byte received).

C.1.2.2 Temperature Display. The system will drive a digital display of temperature.

1. *Physical Interface* The digital display subsystem is connected by a 9600 baud RS-232 connection.

2. *Display Data Format* Display data sent to the temperature display must be an ASCII sequence of the form:

   SOH xxx.y EOR

   where xxx.y is a five character temperature in degree F.

3. *Error Handling* Any character sequence *not* delineated by SOH and EOR will be ignored. Any string so delineated, but *not* of the form xxx.y will be ignored.

4. *Response Time* The temperature display subsystem is fast enough to process a continuous stream of display data at 9600 baud.

C.1.2.3 Fan Control. The system will provide a simple ON/OFF control for a cooling fan.

1. *Physical Interface* The fan will be controlled by the least significant bit (LSB) of a latching TTL parallel port.

2. *Output Format* Writing an odd number (LSB = 1) to this port will turn the fan on. Writing an even number (LSB = 0) to this port will turn the fan off.

C.1.2.4 Graphics Display. The system will drive a "smart" graphics display.

1. *Physical Interface* The graphics display will be connected via a 19200 baud RS-232 full duplex link.

2. *Graphics Command Format* Graphics commands consist of variable length ASCII strings of the format:

   ESC $ < *command* >.

   Detailed commands are listed in the document "Super Kool Graphics Display Model 123 Manual."

C.1.2.5 System Clock. A hardware system clock will be available.

1. *Physical Interface* The clock is readable as a 16-bit parallel port with a 16-bit command and status register.

2. *Clock Resolution* The clock has a resolution of 0.1 seconds.

3. *Clock Format* The clock data format is documented in the specification data sheet "CK-4505 Clock/Calendar Chip Set."

C.1.2.6 User Terminal. The computer operating system provides buffered I/O to the user terminal.

1. *Physical Interface* The keyboard and CRT interfaces are an integral part of the computer system. Access is via defined operating system calls.
2. *Buffered Keyboard Input* Buffered input will provide a string of ASCII characters terminated at the keyboard with a RETURN (ASCII CR). The ASCII CR will be stored as the last character in the buffer. No additional characters (such as a NULL) will be added. The buffer is limited to 81 characters. Any attempt to type beyond 80 characters will result in the buffer being returned with the first 80 characters plus the ASCII CR.
3. *Buffered CRT Output* ASCII output for the CRT is written to a 2000-character (25 line by 80 character) circular buffer. This will update the CRT as fast as the program can write to it. After the 2000th character, the CRT screen will either scroll or overwrite from the top line, depending on the CRT output mode. See the document "XYZ Computer System Manual" for more detail.

C.1.3 System Functions

C.1.3.1 Monitor Temperature. The system shall periodically read and record internally the time and temperature.

1. *Frequency* Temperature will be recorded at 10 second intervals.
2. *Maximum Amount* At least twenty-four hours of data must be stored.

C.1.3.2 Display Temperature. The system will display the current temperature.

1. *Format* The temperature will be displayed in degree F.
2. *Frequency* The temperature display will be updated every 20 seconds.
3. *Response* The temperature display will be updated within 2 seconds after the latest temperature sample has been read.

C.1.3.3 Control Fan The fan will be turned on or off based on comparison of the latest temperature reading with a setpoint value.

1. *Frequency* The fan condition output will be updated every 10 seconds.
2. *Response* The fan control output update will occur within one second of the latest temperature sample.

3. *Operation* The fan will be turned on if the temperature is greater than or equal to the setpoint to the nearest 0.1 degree F. The fan will be turned off if the temperature is less than the setpoint.
4. *Default Setpoint* The initial (default) temperature setpoint shall be 70 degree F.
5. *Setpoint Change* The setpoint can be set to the nearest 0.01 degree F by a user keyboard command. Appropriate input data integrity checks should insure that an illegal value is not processed.

C.1.3.4 Display temperature Plot. On user command, the temperature will be plotted as a function of time on the graphics display.

1. *Keyboard Command* A user keyboard command will cause a new temperature graph to be displayed. The user keyboard command will specify the start and stop times for the graph. Both start and stop times must be within the past twenty-four hours. Stop time must be later than start time. Invalid commands will be ignored.
2. *Vertical Resolution* The temperature will be plotted to the nearest 1 degree F.
3. *Horizontal Resolution* One hundred points will be plotted across the full horizontal width of the display, between the designated start and stop times.
4. *Response* The entire display will be completed within five seconds of the keyboard RETURN terminating the keyboard command.

C.1.3.5 Process Keyboard Commands. Any keyboard input except the commands specified in sections ?? and C.1.3.4 will be ignored.

C.2 DCDS DataDictionary

```
ALPHA: CALCULATE_GRAPHICS_COORDS.
   DESCRIPTION:
      "Calculate graphics coordinates for a
given temp point.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: T_TEMP_F
      DATA: X_SCALE
      DATA: Y_SCALE.
   OUTPUTS:
      DATA: XCOORD_OUT
      DATA: YCOORD_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      SUBNET: CREATE_PLOT_FILE.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_4_2.
ALPHA: CONVERT_TO_F.
   DESCRIPTION:
      "Converts degrees C to degrees F.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: TEMP_C.
   OUTPUTS:
      DATA: DISPLAY_TEMP_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TEMP_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_2_1.
ALPHA: CONVERT_TO_F_TEMP.
   DESCRIPTION:
      "Converts temp point to degrees F.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: T_TEMP_C.
   OUTPUTS:
      DATA: T_TEMP_F.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      SUBNET: CREATE_PLOT_FILE.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4.
ALPHA: CREATE_GRAPHICS_COMMAND.
   DESCRIPTION:
      "Creates a graphics format command for
plot file.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: GRAPHICS_HEAD_OUT
      DATA: GRAPHICS_TAIL_OUT
      DATA: XCOORD_OUT
      DATA: YCOORD_OUT.
   OUTPUTS:
      DATA: GRAPHICS_COMMAND_OUT
      FILE: PLOT_DATA_OUT.
   DOCUMENTED BY:
      SOURCE: GRAPHICS_DISPLAY_MODEL_123_MANUAL.
   REFERRED BY:
      SUBNET: CREATE_PLOT_FILE.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_4_2.
ALPHA: DETERMINE_MSG_TYPE.
   DESCRIPTION:
      "Dummy module because DCDS wouldn't let me
change an OR-node to a CONSIDER-OR.  Sets
SETPOINT or PLOT to TRUE if COMMAND_TYPE_IN is
that type.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: COMMAND_TYPE_IN.
   OUTPUTS:
      DATA: PLOT
      DATA: SETPOINT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5
      ORIGINATING_REQUIREMENT: B_3_4_1
      ORIGINATING_REQUIREMENT: B_3_5.
ALPHA: DETERMINE_SCALE_FACTOR.
   DESCRIPTION:
      "Determine the X and Y scale factors for
the temperature plot from the start and stop
times and the temperature extremes.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: TEMP_MAX
      DATA: TEMP_MIN
      DATA: TEMP_START_IN
      DATA: TEMP_STOP_IN.
   OUTPUTS:
      DATA: X_SCALE
      DATA: Y_SCALE.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4_2
      ORIGINATING_REQUIREMENT: B_3_4_3.
ALPHA: FORM_FAN_MSG.
   DESCRIPTION:
      "Creates control message to fan.".
   ENTERED_BY:
      "Hartrum".
   FORMS:
      MESSAGE: FAN_MESSAGE_OUT.
   INPUTS:
      DATA: FAN_DATA_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      SUBNET: CONTROL_FAN.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3.
```

```
ALPHA: FORM_PLOT_MESSAGE.
   DESCRIPTION:
      "Creates plot message for display from
plot file.".
   ENTERED_BY:
      "Hartrum".
   FORMS:
      MESSAGE: TEMP_PLOT_OUT.
   INPUTS:
      FILE: PLOT_DATA_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4.
ALPHA: MAKE_DISPL_MESSAGE.
   DESCRIPTION:
      "Creates message to temperature display.".
   ENTERED_BY:
      "Hartrum".
   FORMS:
      MESSAGE: DISPLAY_MESSAGE_OUT.
   INPUTS:
      DATA: DISPLAY_HEAD_OUT
      DATA: DISPLAY_TAIL_OUT
      DATA: DISPLAY_TEMP_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TEMP_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_2_1.
ALPHA: SEND_SETPOINT_ACK.
   DESCRIPTION:
      "Sends setpoint acknowledge prompt to CRT".
   ENTERED_BY:
      "Hartrum".
   FORMS:
      MESSAGE: TERMINAL_CRT_MESSAGE_OUT.
   INPUTS:
      DATA: SETPOINT_ACK.
   OUTPUTS:
      DATA: CRT_STRING_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5.
ALPHA: SEND_TEMP_REQ.
   DESCRIPTION:
      "Send the request sequence to the temp
sensor.".
   ENTERED_BY:
      "Hartrum".
   FORMS:
      MESSAGE: TEMP_REQUEST_OUT.
   INPUTS:
      DATA: REQUEST_SEQUENCE_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TIME_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1_1.
ALPHA: SET_FAN_OFF.
   DESCRIPTION:
      "Set fan control to OFF (0).".
   ENTERED_BY:
      "Hartrum".
   OUTPUTS:
      DATA: FAN_DATA_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      SUBNET: CONTROL_FAN.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_3_2.
ALPHA: SET_FAN_ON.
   DESCRIPTION:
      "Sets fan control to ON.".
   ENTERED_BY:
      "Hartrum".
   OUTPUTS:
      DATA: FAN_DATA_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      SUBNET: CONTROL_FAN.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_3_2.
ALPHA: SET_NOW_TIME.
   DESCRIPTION:
      "Updates the clock value from the real-
time interrupt.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: SYS_TIME_IN.
   OUTPUTS:
      DATA: NOW_TIME.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TIME_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_5_1
ALPHA: STORE_SETPOINT.
   DESCRIPTION:
      "Update the current temperature
setpoint.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: SETPOINT_VALUE_IN.
   OUTPUTS:
      DATA: SETPOINT_VALUE.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5.
ALPHA: STORE_TEMP.
   DESCRIPTION:
      "Stores a time & temp pair.".
   ENTERED_BY:
      "Hartrum".
   CREATES:
```

```
      ENTITY_CLASS: TEMP_POINT.
   INPUTS:
      DATA: NOW_TIME
      DATA: TEMP_C.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TEMP_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_1_2.
ALPHA: UPDATE_TEMP_RANGE.
   DESCRIPTION:
      "For each temp point, compare to max & min
and update them if needed.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: T_TEMP_C.
   OUTPUTS:
      DATA: TEMP_MAX
      DATA: TEMP_MIN.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4_2.
ALPHA: VALIDATE_TEMP_MSG.
   DESCRIPTION:
      "Validates temperature message & dumps
(ignores) invalids.".
   ENTERED_BY:
      "Hartrum".
   INPUTS:
      DATA: REPORT_TEMP_IN.
   OUTPUTS:
      DATA: TEMP_C
      DATA: VALID.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TEMP_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1.



DATA: BAD_DATA_IN.
   DESCRIPTION:
      "Any keyboard input except a legal
command.".
   ENTERED_BY:
      "Hartrum".
   LOCALITY:
      LOCAL.
   TYPE:
      ENUMERATION.
   MAKES:
      MESSAGE: BAD_COMMAND_IN.
   DOCUMENTED BY:
      SOURCE: S88_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_5.
DATA: CLOCK_TIME.
   DESCRIPTION:
      "A PREDEFINED DATA ITEM WHICH IS
INCREMENTED AT THE SAME RATE AS ENGAGEMENT TIME.
EXCEPT FOR ITS INITIAL_VALUE WHICH IS ARBITRARY,
CLOCK_TIME MAY BE REGARDED AS ENGAGEMENT TIME.
IT HAS NO CLOCK ERROR.".
   LOCALITY:
      GLOBAL.
   TYPE:
      REAL.
   UNITS:
      SECONDS.
   USE:
      BOTH.
DATA: COMMAND_TYPE_IN.
   DESCRIPTION:
      "Type of keyboard command:setpoint,plot,
bad.".
   ENTERED_BY:
      "Hartrum".
   LOCALITY:
      LOCAL.
   RANGE:
      "setpoint, plot".
   TYPE:
      ENUMERATION.
   MAKES:
      MESSAGE: BAD_COMMAND_IN
      MESSAGE: PLOT_COMMAND_IN
      MESSAGE: SETPOINT_COMMAND_IN.
   INPUT TO:
      ALPHA: DETERMINE_MSG_TYPE.
DATA: CRT_STRING_OUT.
   DESCRIPTION:
      "Any buffered output string to the
terminal CRT.".
   LOCALITY:
      LOCAL.
   TYPE:
      ENUMERATION.
   MAKES:
      MESSAGE: TERMINAL_CRT_MESSAGE_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   OUTPUT FROM:
      ALPHA: SEND_SETPOINT_ACK.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5
      ORIGINATING_REQUIREMENT: B_3_4_1.
DATA: DISPLAY_HEAD_OUT.
   DESCRIPTION:
      "ASCII SOH to start display string.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      SOH.
   LOCALITY:
      LOCAL.
   TYPE:
      ENUMERATION.
   MAKES:
      MESSAGE: DISPLAY_MESSAGE_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INPUT TO:
```

```
      ALPHA: MAKE_DISPL_MESSAGE.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_2_2.
DATA: DISPLAY_TAIL_OUT.
   DESCRIPTION:
      "End of ASCII sequence to display.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      EOM.
   LOCALITY:
      LOCAL.
   TYPE:
      ENUMERATION.
   MAKES:
      MESSAGE: DISPLAY_MESSAGE_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INPUT TO:
      ALPHA: MAKE_DISPL_MESSAGE.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_2_2.
DATA: DISPLAY_TEMP_OUT.
   DESCRIPTION:
      "ASCII string xxx.y of temp in degrees F
to display.".
   ENTERED_BY:
      "Hartrum".
   LOCALITY:
      LOCAL.
   MAXIMUM_VALUE:
      999.9.
   MINIMUM_VALUE:
      0.0.
   RESOLUTION:
      0.1.
   TYPE:
      REAL.
   UNITS:
      DEG_F.
   MAKES:
      MESSAGE: DISPLAY_MESSAGE_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INPUT TO:
      ALPHA: MAKE_DISPL_MESSAGE.
   OUTPUT FROM:
      ALPHA: CONVERT_TO_F.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_2_2.
DATA: FAN_DATA_OUT.
   DESCRIPTION:
      "ON/OFF control for the fan.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      0.
   LOCALITY:
      LOCAL.
   MAXIMUM_VALUE:
      1.
   MINIMUM_VALUE:
      0.
   TYPE:
      BOOLEAN.
   MAKES:
      MESSAGE: FAN_MESSAGE_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INPUT TO:
      ALPHA: FORM_FAN_MSG.
   OUTPUT FROM:
      ALPHA: SET_FAN_OFF
      ALPHA: SET_FAN_ON.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_3_2.
DATA: FOUND.
   DESCRIPTION:
      "A PREDEFINED DATA ITEM WHICH IS SET TO
EITHER TRUE
      OR FALSE AFTER EACH SELECT ON AN
ENTITY_TYPE OR ENTITY_CLASS.
      FOUND IS SET TO TRUE IF AN INSTANCE
SATISFYING THE SELECTION
      CRITERION IS LOCATED; OTHERWISE, FOUND IS
ASSIGNED THE VALUE
      FALSE.".
   INITIAL_VALUE:
      FALSE.
   LOCALITY:
      LOCAL.
   TYPE:
      BOOLEAN.
   USE:
      BOTH.
DATA: GRAPHICS_COMMAND_OUT.
   DESCRIPTION:
      "A graphics command to the display.".
   ENTERED_BY:
      "Hartrum".
   LOCALITY:
      LOCAL.
   INCLUDES:
      DATA: GRAPHICS_HEAD_OUT
      DATA: GRAPHICS_TAIL_OUT
      DATA: XCOORD_OUT
      DATA: YCOORD_OUT.
   CONTAINED IN:
      FILE: PLOT_DATA_OUT.
   DOCUMENTED BY:
      SOURCE: GRAPHICS_DISPLAY_MODEL_123_MANUAL
      SOURCE: S88_001.
   OUTPUT FROM:
      ALPHA: CREATE_GRAPHICS_COMMAND.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_2_1.
DATA: GRAPHICS_HEAD_OUT.
   DESCRIPTION:
      "ASCII ESC $ header for graphics
command.".
   ENTERED_BY:
      "Hartrum".
   LOCALITY:
      LOCAL.
   TYPE:
      ENUMERATION.
   DOCUMENTED BY:
      SOURCE: GRAPHICS_DISPLAY_MODEL_123_MANUAL.
   INCLUDED IN:
      DATA: GRAPHICS_COMMAND_OUT.
```

```
   INPUT TO:
      ALPHA: CREATE_GRAPHICS_COMMAND.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_2_1.
DATA: GRAPHICS_TAIL_OUT.
   DESCRIPTION:
      "ASCII ! terminating graphics command.".
   ENTERED_BY:
      "Hartrum".
   LOCALITY:
      LOCAL.
   TYPE:
      ENUMERATION.
   DOCUMENTED BY:
      SOURCE: GRAPHICS_DISPLAY_MODEL_123_MANU'L.
   INCLUDED IN:
      DATA: GRAPHICS_COMMAND_OUT.
   INPUT TO:
      ALPHA: CREATE_GRAPHICS_COMMAND.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_4_2.
DATA: NOW_TIME.
   DESCRIPTION:
      "Current time, last value read from
clock.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      0.0.
   LOCALITY:
      GLOBAL.
   MAXIMUM_VALUE:
      99999.9.
   MINIMUM_VALUE:
      0.0.
   RESOLUTION:
      0.1.
   TYPE:
      REAL.
   UNITS:
      SECOND.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INPUT TO:
      ALPHA: STORE_TEMP.
   OUTPUT FROM:
      ALPHA: SET_NOW_TIME.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_5_1.
DATA: PLOT.
   DESCRIPTION:
      "Dummy var to indicate that msg type is
plot.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      FALSE.
   LOCALITY:
      LOCAL.
   TYPE:
      BOOLEAN.
   DOCUMENTED BY:
      SOURCE: S88_001.
   OUTPUT FROM:
      ALPHA: DETERMINE_MSG_TYPE.
   REFERRED BY:
      B_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4_1.
DATA: RECORD_FOUND.
   DESCRIPTION:
      "A PREDEFINED DATA ITEM WHICH IS SET TO
EITHER TRUE OR FALSE AFTER EACH SELECT ON A
FILE IN A BETA OR GAMMA.  RECORD_FOUND IS
SET TO TRUE IF A RECORD SATISFYING THE
SELECTION CRITERION IS LOCATED; OTHERWISE,
RECORD_FOUND IS ASSIGNED THE VALUE
      FALSE.".
   INITIAL_VALUE:
      FALSE.
   LOCALITY:
      LOCAL.
   TYPE:
      BOOLEAN.
   USE:
      BOTH.
DATA: REPORT_HEAD_IN.
   DESCRIPTION:
      "ASCII ESC]T header for temp report.".
   ENTERED_BY:
      "Hartrum".
   LOCALITY:
      LOCAL.
   TYPE:
      ENUMERATION.
   MAKES:
      MESSAGE: TEMP_REPORT_IN.
   DOCUMENTED BY:
      SOURCE: S88_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1_5.
DATA: REPORT_TAIL_IN.
   DESCRIPTION:
      "ASCII $ terminating temp report.".
   ENTERED_BY:
      "Hartrum".
   LOCALITY:
      LOCAL.
   TYPE:
      ENUMERATION.
   MAKES:
      MESSAGE: TEMP_REPORT_IN.
   DOCUMENTED BY:
      SOURCE: S88_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1_5.
DATA: REPORT_TEMP_IN.
   DESCRIPTION:
      "Temperature in degree C from sensor.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      0.0.
   LOCALITY:
      LOCAL.
   MAXIMUM_VALUE:
      999.9.
   MINIMUM_VALUE:
      0.0.
   RESOLUTION:
```

```
      0.1.
    TYPE:
      REAL.
    UNITS:
      DEG_C.
    MAKES:
      MESSAGE: TEMP_REPORT_IN.
    DOCUMENTED BY:
      SOURCE: S88_001.
    INPUT TO:
      ALPHA: VALIDATE_TEMP_MSG.
    TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1_5.
DATA: REQUEST_SEQUENCE_OUT.
    DESCRIPTION:
      "ASCII ESC]R$ to request temp report.".
    ENTERED_BY:
      "Hartrum".
    LOCALITY:
      LOCAL.
    TYPE:
      ENUMERATION.
    MAKES:
      MESSAGE: TEMP_REQUEST_OUT.
    DOCUMENTED BY:
      SOURCE: S88_001.
    INPUT TO:
      ALPHA: SEND_TEMP_REQ.
    TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1_3.
DATA: SETPOINT.
    DESCRIPTION:
      "Dummy variable to reflect if message type
is setpoint.".
    ENTERED_BY:
      "Hartrum".
    INITIAL_VALUE:
      FALSE.
    LOCALITY:
      LOCAL.
    TYPE:
      BOOLEAN.
    DOCUMENTED BY:
      SOURCE: S88_001.
    OUTPUT FROM:
      ALPHA: DETERMINE_MSG_TYPE.
    REFERRED BY:
      SUBNET: CONTROL_FAN
      R_NET: TERM_NET.
    TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5.
DATA: SETPOINT_ACK.
    DESCRIPTION:
      "Acknowledgement sent to crt.".
    ENTERED_BY:
      "Hartrum".
    INITIAL_VALUE:
      NULL.
    LOCALITY:
      LOCAL.
    RANGE:
      "setpoint OK".
    TYPE:
      ENUMERATION.
    DOCUMENTED BY:
      SOURCE: S88_001.
    INPUT TO:
      ALPHA: SEND_SETPOINT_ACK.
    TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5.
DATA: SETPOINT_VALUE.
    DESCRIPTION:
      "Current value of setpoint.".
    ENTERED_BY:
      "Hartrum".
    INITIAL_VALUE:
      70.0.
    LOCALITY:
      GLOBAL.
    MAXIMUM_VALUE:
      999.9.
    MINIMUM_VALUE:
      0.0.
    RESOLUTION:
      0.1.
    TYPE:
      REAL.
    UNITS:
      DEG_F.
    DOCUMENTED BY:
      SOURCE: S88_001.
    OUTPUT FROM:
      ALPHA: STORE_SETPOINT.
    TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_3.
DATA: SETPOINT_VALUE_IN.
    DESCRIPTION:
      "Setpoint value in degree F as entered by
user keyboard.".
    ENTERED_BY:
      "Hartrum".
    INITIAL_VALUE:
      70.0.
    LOCALITY:
      GLOBAL.
    MAXIMUM_VALUE:
      999.9.
    MINIMUM_VALUE:
      0.0.
    RESOLUTION:
      0.1.
    TYPE:
      REAL.
    UNITS:
      DEG_F.
    MAKES:
      MESSAGE: SETPOINT_COMMAND_IN.
    DOCUMENTED BY:
      SOURCE: S88_001.
    INPUT TO:
      ALPHA: STORE_SETPOINT.
    TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5.
DATA: START_HOUR_IN.
    DESCRIPTION:
      "Hour part of temp plot start time.".
    ENTERED_BY:
      "Hartrum".
    INITIAL_VALUE:
      0.
```

```
      LOCALITY:
         LOCAL.
      MAXIMUM_VALUE:
         23.
      MINIMUM_VALUE:
         0.
      RESOLUTION:
         1.
      TYPE:
         INTEGER.
      UNITS:
         HOUR.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INCLUDED IN:
         DATA: TEMP_START_IN.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_4_1.
   DATA: START_MIN_IN.
      DESCRIPTION:
         "Minutes portion of plot start time.".
      ENTERED_BY:
         "Hartrum".
      INITIAL_VALUE:
         0.
      LOCALITY:
         LOCAL.
      MAXIMUM_VALUE:
         59.
      MINIMUM_VALUE:
         0.
      RESOLUTION:
         1.
      TYPE:
         INTEGER.
      UNITS:
         MINUTE.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INCLUDED IN:
         DATA: TEMP_START_IN.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_4_1.
   DATA: START_TIME.
   DATA: STOP_HOUR_IN.
      DESCRIPTION:
         "Hour portion of plot stop time.".
      ENTERED_BY:
         "Hartrum".
      INITIAL_VALUE:
         0.
      LOCALITY:
         LOCAL.
      MAXIMUM_VALUE:
         23.
      MINIMUM_VALUE:
         0.
      RESOLUTION:
         1.
      TYPE:
         INTEGER.
      UNITS:
         HOUR.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INCLUDED IN:
         DATA: TEMP_STOP_IN.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_4_1.
   DATA: STOP_MIN_IN.
      DESCRIPTION:
         "Minute portion of plot stop time.".
      ENTERED_BY:
         "Hartrum".
      INITIAL_VALUE:
         0.
      LOCALITY:
         LOCAL.
      MAXIMUM_VALUE:
         59.
      MINIMUM_VALUE:
         0.
      RESOLUTION:
         1.
      TYPE:
         INTEGER.
      UNITS:
         MINUTE.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INCLUDED IN:
         DATA: TEMP_STOP_IN.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_4_1.
   DATA: SYS_TIME_IN.
      DESCRIPTION:
         "Current system time, resolution 0.1s.".
      ENTERED_BY:
         "Hartrum".
      INITIAL_VALUE:
         0.0.
      LOCALITY:
         GLOBAL.
      MAXIMUM_VALUE:
         1048576.
      MINIMUM_VALUE:
         0.
      RESOLUTION:
         100.
      TYPE:
         INTEGER.
      UNITS:
         MS.
      MAKES:
         MESSAGE: TIME_MESSAGE_IN.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INPUT TO:
         ALPHA: SET_NOW_TIME.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_2_5_3.
   DATA: TEMP_C.
      DESCRIPTION:
         "Latest temperature reading.".
      ENTERED_BY:
         "Hartrum".
      INITIAL_VALUE:
         0.0.
      LOCALITY:
         GLOBAL.
```

```
      MAXIMUM_VALUE:
         999.9.
      MINIMUM_VALUE:
         0.0.
      RESOLUTION:
         0.1.
      TYPE:
         REAL.
      UNITS:
         DEG_C.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INPUT TO:
         ALPHA: CONVERT_TO_F
         ALPHA: STORE_TEMP.
      OUTPUT FROM:
         ALPHA: VALIDATE_TEMP_MSG.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_1_2.
DATA: TEMP_F.
      REFERRED BY:
         SUBNET: CONTROL_FAN.
DATA: TEMP_MAX.
      INPUT TO:
         ALPHA: DETERMINE_SCALE_FACTOR.
      OUTPUT FROM:
         ALPHA: UPDATE_TEMP_RANGE.
DATA: TEMP_MIN.
      INPUT TO:
         ALPHA: DETERMINE_SCALE_FACTOR.
      OUTPUT FROM:
         ALPHA: UPDATE_TEMP_RANGE.
DATA: TEMP_START_IN.
      DESCRIPTION:
         "Start time for temperature plot, input by
user.".
      ENTERED_BY:
         "Hartrum".
      INCLUDES:
         DATA: START_HOUR_IN
         DATA: START_MIN_IN.
      MAKES:
         MESSAGE: PLOT_COMMAND_IN.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INPUT TO:
         ALPHA: DETERMINE_SCALE_FACTOR.
      REFERRED BY:
         R_NET: TERM_NET.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_4_1.
DATA: TEMP_STOP_IN.
      DESCRIPTION:
         "End time for temperature plot.".
      ENTERED_BY:
         "Hartrum".
      INCLUDES:
         DATA: STOP_HOUR_IN
         DATA: STOP_MIN_IN.
      MAKES:
         MESSAGE: PLOT_COMMAND_IN.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INPUT TO:
         ALPHA: DETERMINE_SCALE_FACTOR.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_4_1.
DATA: T_TEMP_C.
      DESCRIPTION:
         "A stored temperature point.".
      ENTERED_BY:
         "Hartrum".
      INITIAL_VALUE:
         0.0.
      LOCALITY:
         LOCAL.
      MAXIMUM_VALUE:
         999.9.
      MINIMUM_VALUE:
         0.0.
      RESOLUTION:
         0.1.
      TYPE:
         REAL.
      UNITS:
         DEG_C.
      ASSOCIATED WITH:
         ENTITY_CLASS: TEMP_POINT.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INPUT TO:
         ALPHA: CONVERT_TO_F_TEMP
         ALPHA: UPDATE_TEMP_RANGE.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_1_2.
DATA: T_TEMP_F.
      DESCRIPTION:
         "Fahrenheit temp for each temp data
point.".
      ENTERED_BY:
         "Hartrum".
      INITIAL_VALUE:
         0.0.
      LOCALITY:
         GLOBAL.
      MAXIMUM_VALUE:
         999.9.
      MINIMUM_VALUE:
         0.0.
      RESOLUTION:
         0.1.
      TYPE:
         REAL.
      UNITS:
         DEF_F.
      DOCUMENTED BY:
         SOURCE: S88_001.
      INPUT TO:
         ALPHA: CALCULATE_GRAPHICS_COORDS.
      OUTPUT FROM:
         ALPHA: CONVERT_TO_F_TEMP.
      TRACED FROM:
         ORIGINATING_REQUIREMENT: B_3_4.
DATA: T_TIME.
      DESCRIPTION:
         "A time at which a temperature is
stored.".
      ENTERED_BY:
         "Hartrum".
      INITIAL_VALUE:
```

```
      0.0.
   LOCALITY:
      LOCAL.
   MAXIMUM_VALUE:
      99999.9.
   MINIMUM_VALUE:
      0.0.
   RESOLUTION:
      0.1.
   TYPE:
      REAL.
   UNITS:
      SECONDS.
   ASSOCIATED WITH:
      ENTITY_CLASS: TEMP_POINT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_1_2.
DATA: VALID.
   DESCRIPTION:
      "Flag indicating validity of temp
message.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      FALSE.
   LOCALITY:
      LOCAL.
   TYPE:
      BOOLEAN.
   DOCUMENTED BY:
      SOURCE: S88_001.
   OUTPUT FROM:
      ALPHA: VALIDATE_TEMP_MSG.
   REFERRED BY:
      R_NET: TEMP_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1_5.
DATA: XCOORD_OUT.
   DESCRIPTION:
      "X coord (1..640) in device coordinates to
display.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      0.
   LOCALITY:
      LOCAL.
   MAXIMUM_VALUE:
      640.
   MINIMUM_VALUE:
      0.
   RESOLUTION:
      1.
   TYPE:
      INTEGER.
   UNITS:
      PIXEL.
   ORDERS:
      FILE: PLOT_DATA_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCLUDED IN:
      DATA: GRAPHICS_COMMAND_OUT.
   INPUT TO:
      ALPHA: CREATE_GRAPHICS_COMMAND.
   OUTPUT FROM:
      ALPHA: CALCULATE_GRAPHICS_COORDS.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4_3.
DATA: X_SCALE.
   INPUT TO:
      ALPHA: CALCULATE_GRAPHICS_COORDS.
   OUTPUT FROM:
      ALPHA: DETERMINE_SCALE_FACTOR.
DATA: YCOORD_OUT.
   DESCRIPTION:
      "Y coordinate (1..300) in device
coordinates for display.".
   ENTERED_BY:
      "Hartrum".
   INITIAL_VALUE:
      0.
   LOCALITY:
      LOCAL.
   MAXIMUM_VALUE:
      300.
   MINIMUM_VALUE:
      0.
   RESOLUTION:
      1.
   TYPE:
      INTEGER.
   UNITS:
      PIXEL.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCLUDED IN:
      DATA: GRAPHICS_COMMAND_OUT.
   INPUT TO:
      ALPHA: CREATE_GRAPHICS_COMMAND.
   OUTPUT FROM:
      ALPHA: CALCULATE_GRAPHICS_COORDS.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4_2.
DATA: Y_SCALE.
   INPUT TO:
      ALPHA: CALCULATE_GRAPHICS_COORDS.
   OUTPUT FROM:
      ALPHA: DETERMINE_SCALE_FACTOR.



DECISION: PLOT_COMMAND.
   ALTERNATIVES:
      "1. Single command line including start &
stop times.
         2. Single input command, then prompt
user to enter start and stop times.".
   CHOICE:
      " Alternative 1.".
   ENTERED_BY:
      "Hartrum".
   PROBLEM:
      "Format & protocol of temperature plot
command.".
```

```
    TRACES TO:
       MESSAGE: PLOT_COMMAND_IN.
    DOCUMENTED BY:
       SOURCE: S88_001.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_3_4_1.
DECISION: SETPOINT_COMMAND.
    ALTERNATIVES:
       "1. Single command string including
setpoint value.
        2. Enter command and prompt for
value.".
    CHOICE:
       "Alternative 1.".
    ENTERED_BY:
       "Hartrum".
    PROBLEM:
       "Format and protocol of setpoint
command.".
    TRACES TO:
       MESSAGE: SETPOINT_COMMAND_IN.
    DOCUMENTED BY:
       SOURCE: S88_001.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_3_3_5.
ENTITY_CLASS: TEMP_POINT.
    DESCRIPTION:
       "A log of an individual temperature
point.".
    ENTERED_BY:
       "Hartrum".
    ASSOCIATES:
       DATA: T_TEMP_C
       DATA: T_TIME.
    CREATED BY:
       ALPHA: STORE_TEMP.
    REFERRED BY:
       R_NET: TERM_NET.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_3_1_2.
FILE: PLOT_DATA_OUT.
    DESCRIPTION:
       "Data for a full temperature plot sent to
graphics.".
    ENTERED_BY:
       "Hartrum".
    LOCALITY:
       LOCAL.
    CONTAINS:
       DATA: GRAPHICS_COMMAND_OUT.
    MAKES:
       MESSAGE: TEMP_PLOT_OUT.
    DOCUMENTED BY:
       SOURCE: S88_001.
    INPUT TO:
       ALPHA: FORM_PLOT_MESSAGE.
    ORDERED BY:
       DATA: XCOORD_OUT.
    OUTPUT FROM:
       ALPHA: CREATE_GRAPHICS_COMMAND.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_3_4.
INPUT_INTERFACE: FROM_TEMP.
    DESCRIPTION:
       "Receives temperature reports from
sensor.".
    ENTERED_BY:
       "Hartrum".
    CONNECTS TO:
       SUBSYSTEM: TEMP_SENSOR.
    ENABLES:
       R_NET: TEMP_NET.
    PASSES:
       MESSAGE: TEMP_REPORT_IN.
    DOCUMENTED BY:
       SOURCE: S88_001.
    REFERRED BY:
       R_NET: TEMP_NET.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_2_1
       ORIGINATING_REQUIREMENT: B_2_1_5.
INPUT_INTERFACE: FROM_TERMINAL.
    DESCRIPTION:
       "Receives keyboard input from the user's
terminal.".
    ENTERED_BY:
       "Hartrum".
    CONNECTS TO:
       SUBSYSTEM: TERMINAL.
    ENABLES:
       R_NET: TERM_NET.
    PASSES:
       MESSAGE: BAD_COMMAND_IN
       MESSAGE: PLOT_COMMAND_IN
       MESSAGE: SETPOINT_COMMAND_IN.
    DOCUMENTED BY:
       SOURCE: S88_001.
    REFERRED BY:
       R_NET: TERM_NET.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_3_3_5
       ORIGINATING_REQUIREMENT: B_3_4_1
       ORIGINATING_REQUIREMENT: B_3_5.
INPUT_INTERFACE: FROM_TIME.
    DESCRIPTION:
       "System timer, interrupt driven, 0.1s
resolution.".
    ENTERED_BY:
       "Hartrum".
    CONNECTS TO:
       SUBSYSTEM: SYSTEM_CLOCK.
    ENABLES:
       R_NET: TIME_NET.
    PASSES:
       MESSAGE: TIME_MESSAGE_IN.
    DOCUMENTED BY:
       SOURCE: S88_001.
    REFERRED BY:
       R_NET: TIME_NET.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_2_5_1.
MESSAGE: BAD_COMMAND_IN.
    DESCRIPTION:
       "A meaningless terminal keyboard input.".
    ENTERED_BY:
       "Hartrum".
    DOCUMENTED BY:
       SOURCE: S88_001.
    MADE BY:
       DATA: BAD_DATA_IN
```

```
      DATA: COMMAND_TYPE_IN.
   PASSED THROUGH:
      INPUT_INTERFACE: FROM_TERMINAL.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_5.
MESSAGE: DISPLAY_MESSAGE_OUT.
   DESCRIPTION:
      "ASCII sequence SOH xxx.y EOR to
display.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   FORMED BY:
      ALPHA: MAKE_DISPL_MESSAGE.
   MADE BY:
      DATA: DISPLAY_HEAD_OUT
      DATA: DISPLAY_TAIL_OUT
      DATA: DISPLAY_TEMP_OUT.
   PASSED THROUGH:
      OUTPUT_INTERFACE: TO_DISPLAY.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_2_2.
MESSAGE: FAN_MESSAGE_OUT.
   DESCRIPTION:
      "TTL bit 0 or 1 for OFF or ON.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   FORMED BY:
      ALPHA: FORM_FAN_MSG.
   MADE BY:
      DATA: FAN_DATA_OUT.
   PASSED THROUGH:
      OUTPUT_INTERFACE: TO_FAN.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_3_2.
MESSAGE: PLOT_COMMAND_IN.
   DESCRIPTION:
      "Command to draw a temperature curve.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   MADE BY:
      DATA: COMMAND_TYPE_IN
      DATA: TEMP_START_IN
      DATA: TEMP_STOP_IN.
   PASSED THROUGH:
      INPUT_INTERFACE: FROM_TERMINAL.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4_1
      DECISION: PLOT_COMMAND.
MESSAGE: SETPOINT_COMMAND_IN.
   DESCRIPTION:
      "Command sequence to set new setpoint
value.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   MADE BY:
      DATA: COMMAND_TYPE_IN
      DATA: SETPOINT_VALUE_IN.
   PASSED THROUGH:
      INPUT_INTERFACE: FROM_TERMINAL.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5
      DECISION: SETPOINT_COMMAND.
MESSAGE: TEMP_PLOT_OUT.
   DESCRIPTION:
      "The sequence of graphics commands to draw
a curve.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   FORMED BY:
      ALPHA: FORM_PLOT_MESSAGE.
   MADE BY:
      FILE: PLOT_DATA_OUT.
   PASSED THROUGH:
      OUTPUT_INTERFACE: TO_GRAPHICS_DISPLAY.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_4.
MESSAGE: TEMP_REPORT_IN.
   DESCRIPTION:
      "ASCII sequence ESC ]Txxx.yyy$ from temp
sensor.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   MADE BY:
      DATA: REPORT_HEAD_IN
      DATA: REPORT_TAIL_IN
      DATA: REPORT_TEMP_IN.
   PASSED THROUGH:
      INPUT_INTERFACE: FROM_TEMP.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1.
MESSAGE: TEMP_REQUEST_OUT.
   DESCRIPTION:
      "ASCII string ESC]R$ to trigger temp
report from sensor.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   FORMED BY:
      ALPHA: SEND_TEMP_REQ.
   MADE BY:
      DATA: REQUEST_SEQUENCE_OUT.
   PASSED THROUGH:
      OUTPUT_INTERFACE: TO_TEMP.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1_1.
MESSAGE: TERMINAL_CRT_MESSAGE_OUT.
   DESCRIPTION:
      "Any buffered ASCII sent to the CRT.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   FORMED BY:
      ALPHA: SEND_SETPOINT_ACK.
   MADE BY:
      DATA: CRT_STRING_OUT.
   PASSED THROUGH:
```

```
      OUTPUT_INTERFACE: TO_TERMINAL.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5
      ORIGINATING_REQUIREMENT: B_3_4_1
      ORIGINATING_REQUIREMENT: B_3_5.
MESSAGE: TIME_MESSAGE_IN.
   DESCRIPTION:
      "Contains current system time.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   MADE BY:
      DATA: SYS_TIME_IN.
   PASSED THROUGH:
      INPUT_INTERFACE: FROM_TIME.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_5_1.



ORIGINATING_REQUIREMENT: 2_1.
ORIGINATING_REQUIREMENT: B_2_1.
   DESCRIPTION:
      "Receive temperature.".
   ENTERED_BY:
      "Hartrum".
   INCORPORATES:
      ORIGINATING_REQUIREMENT: B_2_1_1
      ORIGINATING_REQUIREMENT: B_2_1_3
      ORIGINATING_REQUIREMENT: B_2_1_5.
   TRACES TO:
      INPUT_INTERFACE: FROM_TEMP
      R_NET: TEMP_NET
      MESSAGE: TEMP_REPORT_IN
      ALPHA: VALIDATE_TEMP_MSG.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_2_1_1.
   DESCRIPTION:
      "Provide temperature request.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      ALPHA: SEND_TEMP_REQ
      MESSAGE: TEMP_REQUEST_OUT
      OUTPUT_INTERFACE: TO_TEMP.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_2_1.
ORIGINATING_REQUIREMENT: B_2_1_3.
   DESCRIPTION:
      "Temperature request format.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      DATA: REQUEST_SEQUENCE_OUT
      OUTPUT_INTERFACE: TO_TEMP.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_2_1.
ORIGINATING_REQUIREMENT: B_2_1_5.
   DESCRIPTION:
      "Temperature report format.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      INPUT_INTERFACE: FROM_TEMP
      DATA: REPORT_HEAD_IN
      DATA: REPORT_TAIL_IN
      DATA: REPORT_TEMP_IN
      DATA: VALID.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_2_1.
ORIGINATING_REQUIREMENT: B_2_2.
   DESCRIPTION:
      "Provide temperature display.".
   ENTERED_BY:
      "Hartrum".
   INCORPORATES:
      ORIGINATING_REQUIREMENT: B_2_2_2.
   TRACES TO:
      OUTPUT_INTERFACE: TO_DISPLAY.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_2_2_2.
   DESCRIPTION:
      "Display data format.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      DATA: DISPLAY_HEAD_OUT
      MESSAGE: DISPLAY_MESSAGE_OUT
      DATA: DISPLAY_TAIL_OUT
      DATA: DISPLAY_TEMP_OUT
      OUTPUT_INTERFACE: TO_DISPLAY.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_2_2.
ORIGINATING_REQUIREMENT: B_2_3.
   DESCRIPTION:
      "Provide fan control.".
   ENTERED_BY:
      "Hartrum".
   INCORPORATES:
      ORIGINATING_REQUIREMENT: B_2_3_2.
   TRACES TO:
      OUTPUT_INTERFACE: TO_FAN.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_2_3_2.
   DESCRIPTION:
      "Fan output format.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      DATA: FAN_DATA_OUT
      MESSAGE: FAN_MESSAGE_OUT
      ALPHA: SET_FAN_OFF
      ALPHA: SET_FAN_ON
      OUTPUT_INTERFACE: TO_FAN.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
```

```
      ORIGINATING_REQUIREMENT: B_2_3.
ORIGINATING_REQUIREMENT: B_2_4.
   DESCRIPTION:
      "Provide graphics display.".
   ENTERED_BY:
      "Hartrum".
   INCORPORATES:
      ORIGINATING_REQUIREMENT: B_2_4_2.
   TRACES TO:
      MESSAGE: TEMP_PLOT_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_2_4_2.
   DESCRIPTION:
      "Graphics command format.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      ALPHA: CALCULATE_GRAPHICS_COORDS
      ALPHA: CREATE_GRAPHICS_COMMAND
      DATA: GRAPHICS_TAIL_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_2_4.
ORIGINATING_REQUIREMENT: B_2_5_1.
   DESCRIPTION:
      "Respond to clock interrupt.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      INPUT_INTERFACE: FROM_TIME
      DATA: NOW_TIME
      ALPHA: SET_NOW_TIME
      MESSAGE: TIME_MESSAGE_IN
      R_NET: TIME_NET.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_2_5_3.
   DESCRIPTION:
      "Clock format.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      DATA: SYS_TIME_IN.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_3_1.
   DESCRIPTION:
      "Monitor temperature.".
   ENTERED_BY:
      "Hartrum".
   INCORPORATES:
      ORIGINATING_REQUIREMENT: B_3_1_1
      ORIGINATING_REQUIREMENT: B_3_1_2.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_3_1_1.
   DESCRIPTION:
      "Monitor temperature frequency.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      PERFORMANCE_REQUIREMENT:
TEMP_MONITOR_INTERVAL.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_1.
ORIGINATING_REQUIREMENT: B_3_1_2.
   DESCRIPTION:
      "Store maximum data.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      ALPHA: STORE_TEMP
      DATA: TEMP_C
      ENTITY_CLASS: TEMP_POINT
      DATA: T_TEMP_C
      DATA: T_TIME.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_1.
ORIGINATING_REQUIREMENT: B_3_2.
   DESCRIPTION:
      "Display Temperature.".
   ENTERED_BY:
      "Hartrum".
   INCORPORATES:
      ORIGINATING_REQUIREMENT: B_3_2_1
      ORIGINATING_REQUIREMENT: B_3_2_2
      ORIGINATING_REQUIREMENT: B_3_2_3.
   TRACES TO:
      OUTPUT_INTERFACE: TO_GRAPHICS_DISPLAY.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_3_2_1.
   DESCRIPTION:
      "Provide temperature display format.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      ALPHA: CONVERT_TO_F
      DATA: GRAPHICS_COMMAND_OUT
      DATA: GRAPHICS_HEAD_OUT
      ALPHA: MAKE_DISPL_MESSAGE
      OUTPUT_INTERFACE: TO_GRAPHICS_DISPLAY.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_2.
ORIGINATING_REQUIREMENT: B_3_2_2.
   DESCRIPTION:
      "Display temperature frequency.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      PERFORMANCE_REQUIREMENT:
TEMP_DISPLAY_FREQUENCY.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_2.
ORIGINATING_REQUIREMENT: B_3_2_3.
   DESCRIPTION:
      "Display temperature response time.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
```

```
      PERFORMANCE_REQUIREMENT:
TEMP_DISPLAY_RESPONSE_TIME.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_2.
ORIGINATING_REQUIREMENT: B_3_3.
   DESCRIPTION:
      "Control fan".
   ENTERED_BY:
      "Hartrum".
   INCORPORATES:
      ORIGINATING_REQUIREMENT: B_3_3_1
      ORIGINATING_REQUIREMENT: B_3_3_2
      ORIGINATING_REQUIREMENT: B_3_3_3
      ORIGINATING_REQUIREMENT: B_3_3_4
      ORIGINATING_REQUIREMENT: B_3_3_5.
   TRACES TO:
      SUBSET: CONTROL_FAN
      ALPHA: FORM_FAN_MSG.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_3_3_1.
   DESCRIPTION:
      "Control fan frequency".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      PERFORMANCE_REQUIREMENT:
FAN_CONTROL_FREQUENCY.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_3.
ORIGINATING_REQUIREMENT: B_3_3_2.
   DESCRIPTION:
      "Control fan response time.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      PERFORMANCE_REQUIREMENT:
FAN_CONTROL_RESPONSE_TIME.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_3.
ORIGINATING_REQUIREMENT: B_3_3_3.
   DESCRIPTION:
      "Compare temp to setpoint.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      DATA: SETPOINT_VALUE.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_3.
ORIGINATING_REQUIREMENT: B_3_3_4.
   DESCRIPTION:
      "Default setpoint.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_3.
ORIGINATING_REQUIREMENT: B_3_3_5.
   DESCRIPTION:
      "Allow setpoint change.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      DATA: CRT_STRING_OUT
      ALPHA: DETERMINE_MSG_TYPE
      INPUT_INTERFACE: FROM_TERMINAL
      ALPHA: SEND_SETPOINT_ACK
      DATA: SETPOINT
      DATA: SETPOINT_ACK
      DECISION: SETPOINT_COMMAND
      MESSAGE: SETPOINT_COMMAND_IN
      DATA: SETPOINT_VALUE_IN
      ALPHA: STORE_SETPOINT
      MESSAGE: TERMINAL_CRT_MESSAGE_OUT
      R_NET: TERM_NET
      OUTPUT_INTERFACE: TO_TERMINAL.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_3.
ORIGINATING_REQUIREMENT: B_3_4.
   DESCRIPTION:
      "Display temperature setpoint.".
   ENTERED_BY:
      "Hartrum".
   INCORPORATES:
      ORIGINATING_REQUIREMENT: B_3_4_1
      ORIGINATING_REQUIREMENT: B_3_4_2
      ORIGINATING_REQUIREMENT: B_3_4_3
      ORIGINATING_REQUIREMENT: B_3_4_4.
   TRACES TO:
      ALPHA: CONVERT_TO_F_TEMP
      ALPHA: FORM_PLOT_MESSAGE
      FILE: PLOT_DATA_OUT
      DATA: T_TEMP_F.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: B_3_4_1.
   DESCRIPTION:
      "Process keyboard command.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      DATA: CRT_STRING_OUT
      ALPHA: DETERMINE_MSG_TYPE
      INPUT_INTERFACE: FROM_TERMINAL
      DATA: PLOT
      DECISION: PLOT_COMMAND
      MESSAGE: PLOT_COMMAND_IN
      DATA: START_HOUR_IN
      DATA: START_MIN_IN
      DATA: STOP_HOUR_IN
      DATA: STOP_MIN_IN
      DATA: TEMP_START_IN
      DATA: TEMP_STOP_IN
      MESSAGE: TERMINAL_CRT_MESSAGE_OUT
      R_NET: TERM_NET
      OUTPUT_INTERFACE: TO_TERMINAL.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
```

```
      ORIGINATING_REQUIREMENT: B_3_4.
ORIGINATING_REQUIREMENT: B_3_4_2.
   DESCRIPTION:
      "Vertical resolution.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      ALPHA: DETERMINE_SCALE_FACTOR
      ALPHA: UPDATE_TEMP_RANGE
      DATA: YCOORD_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_4.
ORIGINATING_REQUIREMENT: B_3_4_3.
   DESCRIPTION:
      "Horizontal resolution.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      ALPHA: DETERMINE_SCALE_FACTOR
      DATA: XCOORD_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_4.
ORIGINATING_REQUIREMENT: B_3_4_4.
   DESCRIPTION:
      "Display temperature plot response time.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      PERFORMANCE_REQUIREMENT:
TEMP_PLOT_RESPONSE_TIME.
   DOCUMENTED BY:
      SOURCE: S88_001.
   INCORPORATED IN:
      ORIGINATING_REQUIREMENT: B_3_4.
ORIGINATING_REQUIREMENT: B_3_5.
   DESCRIPTION:
      "Ignore erroneous keyboard commands.".
   ENTERED_BY:
      "Hartrum".
   TRACES TO:
      MESSAGE: BAD_COMMAND_IN
      DATA: BAD_DATA_IN
      ALPHA: DETERMINE_MSG_TYPE
      INPUT_INTERFACE: FROM_TERMINAL
      MESSAGE: TERMINAL_CRT_MESSAGE_OUT
      R_NET: TERM_NET.
   DOCUMENTED BY:
      SOURCE: S88_001.
ORIGINATING_REQUIREMENT: S2_1.
   DESCRIPTION:
      "The system shall periodically read and
record internally the time and temperature.".
   DOCUMENTED BY:
      SOURCE: S88_001.



OUTPUT_INTERFACE: TO_DISPLAY.
   DESCRIPTION:
      "Connects DPS to the temperature display
to allow outputing the current temperature.".
   ENTERED_BY:
      "Hartrum".
   CONNECTS TO:
      SUBSYSTEM: TEMP_DISPLAY.
   PASSES:
      MESSAGE: DISPLAY_MESSAGE_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TEMP_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_2
      ORIGINATING_REQUIREMENT: B_2_2_2.
OUTPUT_INTERFACE: TO_FAN.
   DESCRIPTION:
      "Connects DPS to fan to allow ON/OFF
signal output.".
   ENTERED_BY:
      "Hartrum".
   CONNECTS TO:
      SUBSYSTEM: FAN.
   PASSES:
      MESSAGE: FAN_MESSAGE_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TEMP_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_3
      ORIGINATING_REQUIREMENT: B_2_3_2.
OUTPUT_INTERFACE: TO_GRAPHICS_DISPLAY.
   DESCRIPTION:
      "Allows graphics commands to be sent to
the graphics display in order to draw
temperature plot.".
   ENTERED_BY:
      "Hartrum".
   CONNECTS TO:
      SUBSYSTEM: GRAPHICS_DISPLAY.
   PASSES:
      MESSAGE: TEMP_PLOT_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_2
      ORIGINATING_REQUIREMENT: B_3_2_1.
OUTPUT_INTERFACE: TO_TEMP.
   DESCRIPTION:
      "Connects DPS to the temperature sensor.
Allows temperature requests to be sent to the
temperature sensor.".
   ENTERED_BY:
      "Hartrum".
   CONNECTS TO:
      SUBSYSTEM: TEMP_SENSOR.
   PASSES:
      MESSAGE: TEMP_REQUEST_OUT.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TIME_NET.
   TRACED FROM:
```

```
      ORIGINATING_REQUIREMENT: B_2_1_1
      ORIGINATING_REQUIREMENT: B_2_1_3.
OUTPUT_INTERFACE: TO_TERMINAL.
   DESCRIPTION:
      "Allows text to be sent to the user
terminal's CRT.".
   ENTERED_BY:
      "Hartrum".
   CONNECTS TO:
      SUBSYSTEM: TERMINAL.
   PASSES:
      MESSAGE: TERMINAL_CRT_MESSAGE_OUT.
   DOCUMENTED BY:
      SOURCE: SSS_001.
   REFERRED BY:
      R_NET: TERM_NET.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_5
      ORIGINATING_REQUIREMENT: B_3_4_1.
PERFORMANCE_REQUIREMENT: FAN_CONTROL_FREQUENCY.
   DESCRIPTION:
      "The fan condition output will be updated
every 10 secs.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: SSS_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_1.
PERFORMANCE_REQUIREMENT:
FAN_CONTROL_RESPONSE_TIME.
   DESCRIPTION:
      "The fan control output update will occur
within one second from the latest temperature
sample.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: SSS_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3_2.
PERFORMANCE_REQUIREMENT: TEMP_DISPLAY_FREQUENCY.
   DESCRIPTION:
      "The temperature display will be updated
every 20 seconds.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: SSS_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_2_2.
PERFORMANCE_REQUIREMENT:
TEMP_DISPLAY_RESPONSE_TIME.
   DESCRIPTION:
      "The temperature display will be updated
within 2 seconds after the latest temperature
sample has been read".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: SSS_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_2_3.
PERFORMANCE_REQUIREMENT: TEMP_MONITOR_INTERVAL.
   DESCRIPTION:
      "Temperature will be recorded at 10 second
intervals.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: SSS_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_1_1.
PERFORMANCE_REQUIREMENT: TEMP_PLOT_RESPONSE_TIME.
   DESCRIPTION:
      "The entire display will be completed
within five seconds of the keyboard RETURN
terminating the keyboard command.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTED BY:
      SOURCE: SSS_001.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_4_4.
R_NET: TEMP_NET.
   DESCRIPTION:
      "Processes temperature reports from temp
sensor.".
   ENTERED_BY:
      "Hartrum".
   REFERS TO:
      SUBNET: CONTROL_FAN
      ALPHA: CONVERT_TO_F
      INPUT_INTERFACE: FROM_TEMP
      ALPHA: MAKE_DISPL_MESSAGE
      ALPHA: STORE_TEMP
      OUTPUT_INTERFACE: TO_DISPLAY
      OUTPUT_INTERFACE: TO_FAN
      DATA: VALID
      ALPHA: VALIDATE_TEMP_MSG.
   ENABLED BY:
      INPUT_INTERFACE: FROM_TEMP.
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_2_1.
   STRUCTURE:
     INPUT_INTERFACE FROM_TEMP
     ALPHA VALIDATE_TEMP_MSG
     IF
        ( VALID )
        DO
           ALPHA CONVERT_TO_F
           ALPHA MAKE_DISPL_MESSAGE
           OUTPUT_INTERFACE TO_DISPLAY
        AND
           ALPHA STORE_TEMP
           TERMINATE
        AND
           SUBNET CONTROL_FAN
           OUTPUT_INTERFACE TO_FAN
        END
     OTHERWISE
        TERMINATE
     END.
R_NET: TERM_NET.
   DESCRIPTION:
      "This r_net processes all messages from
the user console.".
   ENTERED_BY:
      "Hartrum".
```

```
    REFERS TO:
       SUBNET: CREATE_PLOT_FILE
       ALPHA: DETERMINE_MSG_TYPE
       ALPHA: DETERMINE_SCALE_FACTOR
       ALPHA: FORM_PLOT_MESSAGE
       INPUT_INTERFACE: FROM_TERMINAL
       DATA: PLOT
       ALPHA: SEND_SETPOINT_ACK
       DATA: SETPOINT
       ALPHA: STORE_SETPOINT
       ENTITY_CLASS: TEMP_POINT
       DATA: TEMP_START_IN
       OUTPUT_INTERFACE: TO_GRAPHICS_DISPLAY
       OUTPUT_INTERFACE: TO_TERMINAL
       DATA: T_TIME
       ALPHA: UPDATE_TEMP_RANGE.
    DOCUMENTED BY:
       SOURCE: S88_001.
    ENABLED BY:
       INPUT_INTERFACE: FROM_TERMINAL.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_3_3_5
       ORIGINATING_REQUIREMENT: B_3_4_1
       ORIGINATING_REQUIREMENT: B_3_5.
    STRUCTURE:
      INPUT_INTERFACE FROM_TERMINAL
      ALPHA DETERMINE_MSG_TYPE
      IF
         ( SETPOINT )
         ALPHA STORE_SETPOINT
         ALPHA SEND_SETPOINT_ACK
         OUTPUT_INTERFACE TO_TERMINAL
      OR
         ( PLOT )
         SELECT ENTITY_CLASS TEMP_POINT
          SUCH THAT
            ( T_TIME>TEMP_START_IN )
         FOR EACH ENTITY_CLASS TEMP_POINT
          DO
            ALPHA UPDATE_TEMP_RANGE
          END
         ALPHA DETERMINE_SCALE_FACTOR
         FOR EACH ENTITY_CLASS TEMP_POINT
          DO
            SUBNET CREATE_PLOT_FILE
          END
         ALPHA FORM_PLOT_MESSAGE
         OUTPUT_INTERFACE TO_GRAPHICS_DISPLAY
      OTHERWISE
      END
    END.
R_NET: TIME_NET.
    DESCRIPTION:
       "Responds to real-time clock interrupts.".
    ENTERED_BY:
       "Hartrum".
    REFERS TO:
       INPUT_INTERFACE: FROM_TIME
       ALPHA: SEND_TEMP_REQ
       ALPHA: SET_NOW_TIME
       OUTPUT_INTERFACE: TO_TEMP.
    DOCUMENTED BY:
       SOURCE: S88_001.
    ENABLED BY:
       INPUT_INTERFACE: FROM_TIME.
    TRACED FROM:
       ORIGINATING_REQUIREMENT: B_2_5_1.
    STRUCTURE:
      INPUT_INTERFACE FROM_TIME
      DO
         ALPHA SET_NOW_TIME
         TERMINATE
      AND
         ALPHA SEND_TEMP_REQ
         OUTPUT_INTERFACE TO_TEMP
      END
    END.


SOURCE: CK_4505_CLOCK_SPEC_SHEET.
    DESCRIPTION:
       "This is the specification sheet for the
CK-4505 clock/calander chip set, and contains
the clock data formats.".
    ENTERED_BY:
       "Hartrum".
    DOCUMENTS:
       SUBSYSTEM: SYSTEM_CLOCK.
SOURCE: GRAPHICS_DISPLAY_MODEL_123_MANUAL.
    DESCRIPTION:
       "This is the user manual for the graphics
display to be used, and contains the graphics
control sequences.".
    ENTERED_BY:
       "Hartrum".
    DOCUMENTS:
       ALPHA: CREATE_GRAPHICS_COMMAND
       DATA: GRAPHICS_COMMAND_OUT
       DATA: GRAPHICS_HEAD_OUT
       DATA: GRAPHICS_TAIL_OUT.
SOURCE: GRAPHICS_DISPLAY_SYSTEM_MANUAL.
    DOCUMENTS:
       SUBSYSTEM: GRAPHICS_DISPLAY.
SOURCE: S88_001.
    DESCRIPTION:
       "This is the overall source requirement
document for the temperature controller
software.".
    ENTERED_BY:
       "Hartrum".
    DOCUMENTS:
       MESSAGE: BAD_COMMAND_IN
       DATA: BAD_DATA_IN
       ORIGINATING_REQUIREMENT: B_2_1
       ORIGINATING_REQUIREMENT: B_2_1_1
       ORIGINATING_REQUIREMENT: B_2_1_3
       ORIGINATING_REQUIREMENT: B_2_1_5
       ORIGINATING_REQUIREMENT: B_2_2
       ORIGINATING_REQUIREMENT: B_2_2_2
       ORIGINATING_REQUIREMENT: B_2_3
       ORIGINATING_REQUIREMENT: B_2_3_2
       ORIGINATING_REQUIREMENT: B_2_4
       ORIGINATING_REQUIREMENT: B_2_4_2
       ORIGINATING_REQUIREMENT: B_2_5_1
       ORIGINATING_REQUIREMENT: B_2_5_3
       ORIGINATING_REQUIREMENT: B_3_1
       ORIGINATING_REQUIREMENT: B_3_1_1
       ORIGINATING_REQUIREMENT: B_3_1_2
```

```
        ORIGINATING_REQUIREMENT: R_3_2
        ORIGINATING_REQUIREMENT: R_3_2_1
        ORIGINATING_REQUIREMENT: R_3_2_2
        ORIGINATING_REQUIREMENT: R_3_2_3
        ORIGINATING_REQUIREMENT: R_3_3
        ORIGINATING_REQUIREMENT: R_3_3_1
        ORIGINATING_REQUIREMENT: R_3_3_2
        ORIGINATING_REQUIREMENT: R_3_3_3
        ORIGINATING_REQUIREMENT: R_3_3_4
        ORIGINATING_REQUIREMENT: R_3_3_5
        ORIGINATING_REQUIREMENT: R_3_4
        ORIGINATING_REQUIREMENT: R_3_4_1
        ORIGINATING_REQUIREMENT: R_3_4_2
        ORIGINATING_REQUIREMENT: R_3_4_3
        ORIGINATING_REQUIREMENT: R_3_4_4
        ORIGINATING_REQUIREMENT: R_3_5
        ALPHA: CALCULATE_GRAPHICS_COORDS
        SUBNET: CONTROL_FAN
        ALPHA: CONVERT_TO_F
        ALPHA: CONVERT_TO_F_TEMP
        SUBNET: CREATE_PLOT_FILE
        DATA: CRT_STRING_OUT
        ALPHA: DETERMINE_MSG_TYPE
        ALPHA: DETERMINE_SCALE_FACTOR
        DATA: DISPLAY_HEAD_OUT
        MESSAGE: DISPLAY_MESSAGE_OUT
        DATA: DISPLAY_TAIL_OUT
        DATA: DISPLAY_TEMP_OUT
        SUBSYSTEM: FAN
        PERFORMANCE_REQUIREMENT:
FAN_CONTROL_FREQUENCY
        PERFORMANCE_REQUIREMENT:
FAN_CONTROL_RESPONSE_TIME
        DATA: FAN_DATA_OUT
        MESSAGE: FAN_MESSAGE_OUT
        ALPHA: FORM_FAN_MSG
        ALPHA: FORM_PLOT_MESSAGE
        INPUT_INTERFACE: FROM_TEMP
        INPUT_INTERFACE: FROM_TERMINAL
        INPUT_INTERFACE: FROM_TIME
        DATA: GRAPHICS_COMMAND_OUT
        SUBSYSTEM: GRAPHICS_DISPLAY
        ALPHA: MAKE_DISPL_MESSAGE
        DATA: NOW_TIME
        DATA: PLOT
        DECISION: PLOT_COMMAND
        MESSAGE: PLOT_COMMAND_IN
        FILE: PLOT_DATA_OUT
        DATA: REPORT_HEAD_IN
        DATA: REPORT_TAIL_IN
        DATA: REPORT_TEMP_IN
        DATA: REQUEST_SEQUENCE_OUT
        ORIGINATING_REQUIREMENT: S2_1
        ALPHA: SEND_SETPOINT_ACK
        ALPHA: SEND_TEMP_REQ
        DATA: SETPOINT
        DATA: SETPOINT_ACK
        DECISION: SETPOINT_COMMAND
        MESSAGE: SETPOINT_COMMAND_IN
        DATA: SETPOINT_VALUE
        DATA: SETPOINT_VALUE_IN
        ALPHA: SET_FAN_OFF
        ALPHA: SET_FAN_ON
        ALPHA: SET_NOW_TIME
        DATA: START_HOUR_IN
        DATA: START_MIN_IN
        DATA: STOP_HOUR_IN
        DATA: STOP_MIN_IN
        ALPHA: STORE_SETPOINT
        ALPHA: STORE_TEMP
        SUBSYSTEM: SYSTEM_CLOCK
        DATA: SYS_TIME_IN
        DATA: TEMP_C
        SUBSYSTEM: TEMP_DISPLAY
        PERFORMANCE_REQUIREMENT:
TEMP_DISPLAY_FREQUENCY
        PERFORMANCE_REQUIREMENT:
TEMP_DISPLAY_RESPONSE_TIME
        PERFORMANCE_REQUIREMENT:
TEMP_MONITOR_INTERVAL
        MESSAGE: TEMP_PLOT_OUT
        PERFORMANCE_REQUIREMENT:
TEMP_PLOT_RESPONSE_TIME
        MESSAGE: TEMP_REPORT_IN
        MESSAGE: TEMP_REQUEST_OUT
        SUBSYSTEM: TEMP_SENSOR
        DATA: TEMP_START_IN
        DATA: TEMP_STOP_IN
        SUBSYSTEM: TERMINAL
        MESSAGE: TERMINAL_CRT_MESSAGE_OUT
        R_NET: TERM_NET
        MESSAGE: TIME_MESSAGE_IN
        R_NET: TIME_NET
        OUTPUT_INTERFACE: TO_DISPLAY
        OUTPUT_INTERFACE: TO_FAN
        OUTPUT_INTERFACE: TO_GRAPHICS_DISPLAY
        OUTPUT_INTERFACE: TO_TEMP
        OUTPUT_INTERFACE: TO_TERMINAL
        DATA: T_TEMP_C
        DATA: T_TEMP_F
        DATA: T_TIME
        ALPHA: UPDATE_TEMP_RANGE
        DATA: VALID
        ALPHA: VALIDATE_TEMP_MSG
        DATA: XCOORD_OUT
        DATA: YCOORD_OUT.
SOURCE: XYZ_COMPUTER_SYSTEM_MANUAL.
   DESCRIPTION:
      "This is the system manual for the
computer to run the temperature controller
software.  It defines the CRT control modes.".
   ENTERED_BY:
      "Hartrum".
   DOCUMENTS:
      SUBSYSTEM: TERMINAL.
SUBNET: CONTROL_FAN.
   DESCRIPTION:
      "This subnet provides fan control.".
   ENTERED_BY:
      "Hartrum".
   REFERS TO:
      ALPHA: FORM_FAN_MSG
      DATA: SETPOINT
      ALPHA: SET_FAN_OFF
      ALPHA: SET_FAN_ON
      DATA: TEMP_F.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TEMP_NET.
```

```
   TRACED FROM:
      ORIGINATING_REQUIREMENT: B_3_3.
   STRUCTURE:
     IF
        ( TEMP_F>SETPOINT )
        ALPHA SET_FAN_ON
     OTHERWISE
        ALPHA SET_FAN_OFF
     END
     ALPHA FORM_FAN_MSG
     RETURN
   END.
SUBNET: CREATE_PLOT_FILE.
   DESCRIPTION:
      "This subnet generates records in the plot
file.".
   ENTERED_BY:
      "Hartrum".
   REFERS TO:
      ALPHA: CALCULATE_GRAPHICS_COORDS
      ALPHA: CONVERT_TO_F_TEMP
      ALPHA: CREATE_GRAPHICS_COMMAND.
   DOCUMENTED BY:
      SOURCE: S88_001.
   REFERRED BY:
      R_NET: TERM_NET.
   STRUCTURE:
     ALPHA CONVERT_TO_F
     ALPHA CALCULATE_GRAPHICS_COORDS
     ALPHA CREATE_GRAPHICS_COMMAND
     RETURN
   END.
SUBSYSTEM: FAN.
   DESCRIPTION:
      "This is a fan to cool the room on
command.".
   ENTERED_BY:
      "Hartrum".
   CONNECTED TO:
      OUTPUT_INTERFACE: TO_FAN.
   DOCUMENTED BY:
      SOURCE: S88_001.
SUBSYSTEM: GRAPHICS_DISPLAY.
   DESCRIPTION:
      "This is a smart graphics terminal to
display a plot of
       temperature vs. time upon command.".
   ENTERED_BY:
      "Hartrum".
   CONNECTED TO:
      OUTPUT_INTERFACE: TO_GRAPHICS_DISPLAY.
   DOCUMENTED BY:
      SOURCE: GRAPHICS_DISPLAY_SYSTEM_MANUAL
      SOURCE: S88_001.
SUBSYSTEM: SYSTEM_CLOCK.
   DESCRIPTION:
      "This is a system clock that interrupts
the software
       and can be read by the software.".
   ENTERED_BY:
      "Hartrum".
   CONNECTED TO:
      INPUT_INTERFACE: FROM_TIME.
   DOCUMENTED BY:
      SOURCE: CK_4506_CLOCK_SPEC_SHEET
      SOURCE: S88_001.
SUBSYSTEM: TEMP_DISPLAY.
   DESCRIPTION:
      "This is a digital display to display the
current
       temperature in degrees F.".
   ENTERED_BY:
      "Hartrum".
   CONNECTED TO:
      OUTPUT_INTERFACE: TO_DISPLAY.
   DOCUMENTED BY:
      SOURCE: S88_001.
SUBSYSTEM: TEMP_SENSOR.
   DESCRIPTION:
      "This temperature sensor provides
temperature readings in
       response to request message.".
   ENTERED_BY:
      "Hartrum".
   CONNECTED TO:
      INPUT_INTERFACE: FROM_TEMP
      OUTPUT_INTERFACE: TO_TEMP.
   DOCUMENTED BY:
      SOURCE: S88_001.
SUBSYSTEM: TERMINAL.
   DESCRIPTION:
      "This is the user terminal used for user
I/O.".
   ENTERED_BY:
      "Hartrum".
   CONNECTED TO:
      INPUT_INTERFACE: FROM_TERMINAL
      OUTPUT_INTERFACE: TO_TERMINAL.
   DOCUMENTED BY:
      SOURCE: S88_001
      SOURCE: XYZ_COMPUTER_SYSTEM_MANUAL.
```

C.3 DCDS Graphic R-Nets and Subnets

The R-Nets and Subnets for the Temperature Controller are provided in the following figures.

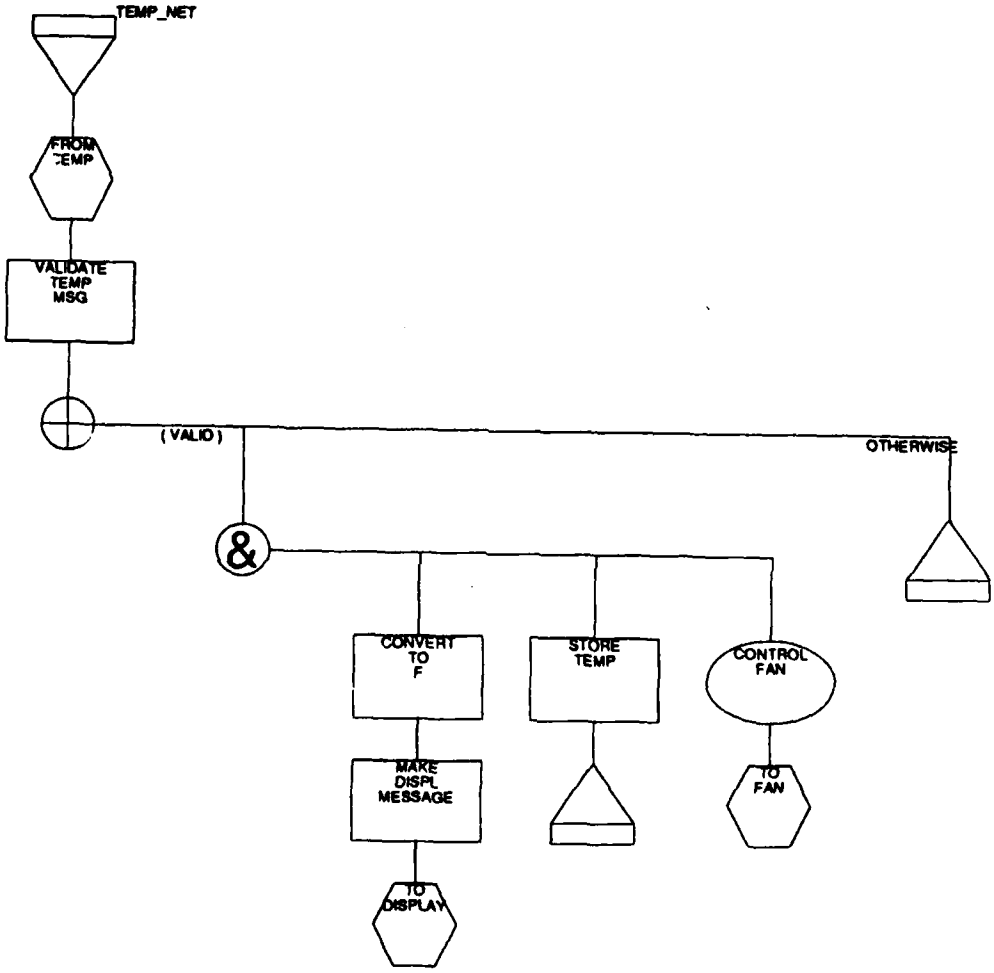


Figure C.1. R_net diagram for TEMP_NET

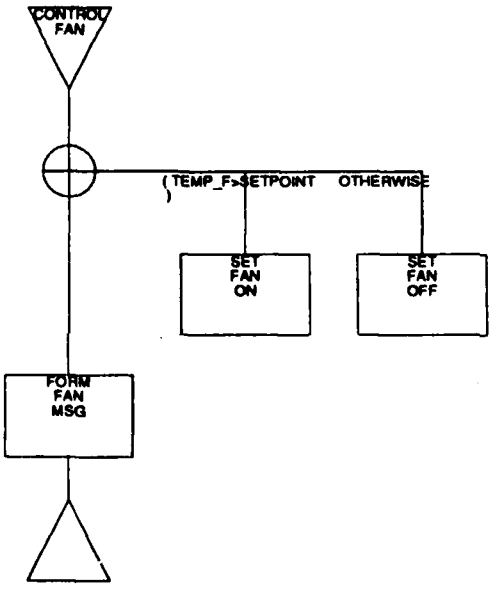


Figure C.2. Subnet diagram for CONTROL_FAN

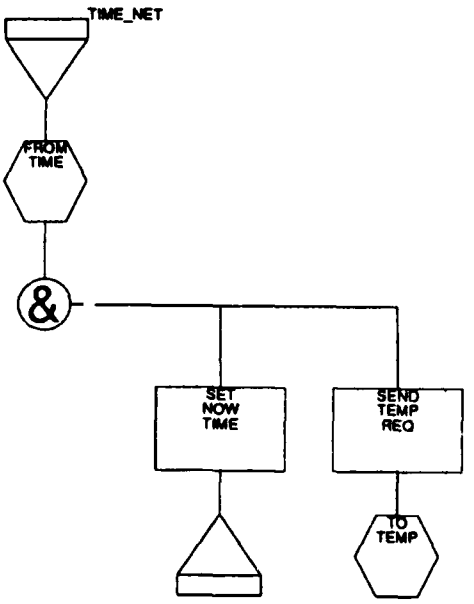


Figure C.3. R_net diagram for TIME_NET

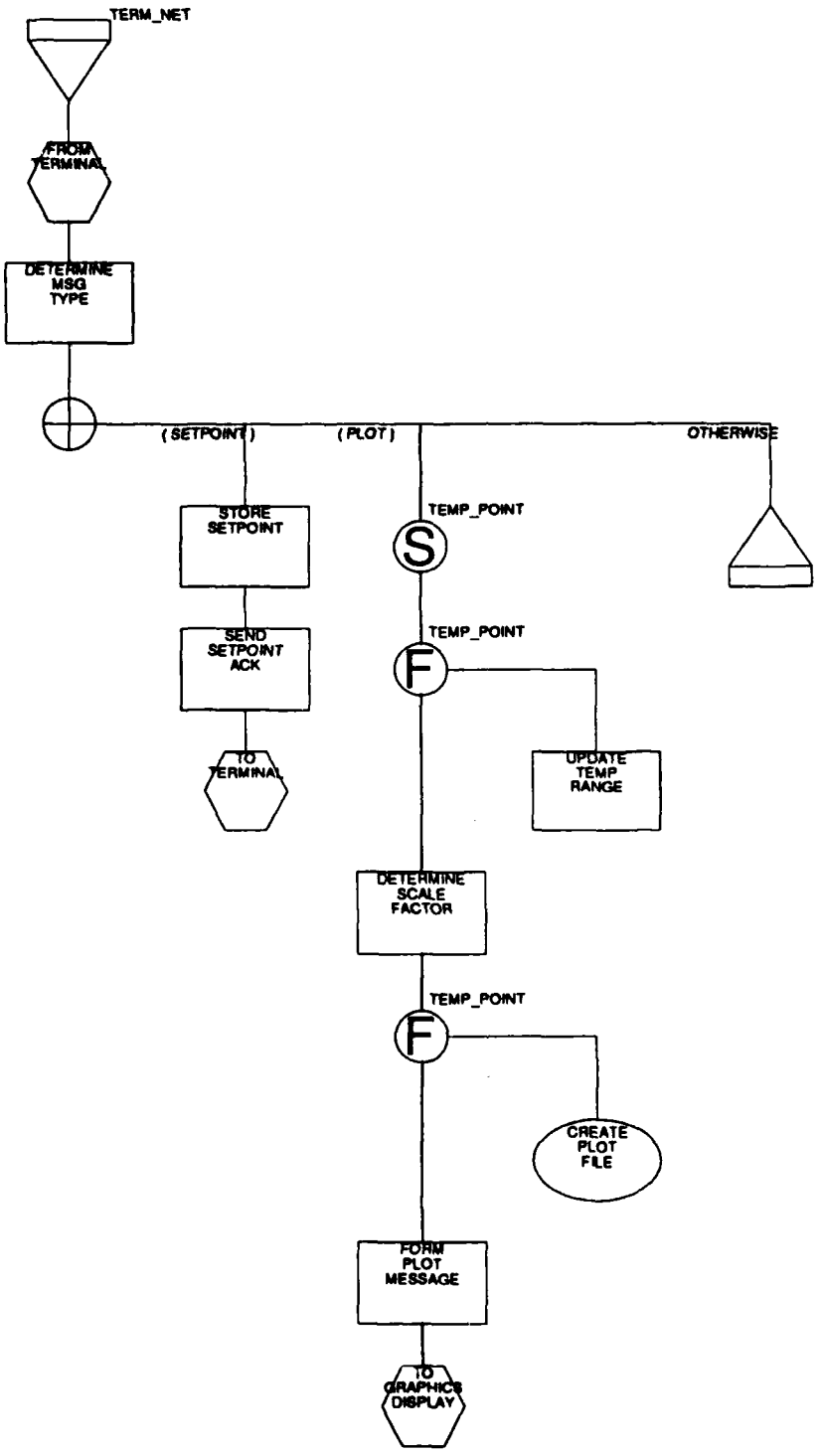


Figure C.4. R_net diagram for TERM_NET

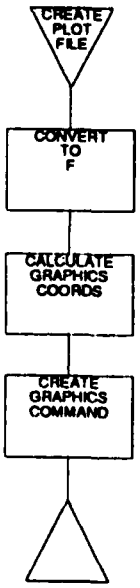


Figure C.5. Subnet diagram for CREATE_PLOT_FILE

Vita

Captain Patrick Denis Barnes was born April 16, 1955, in St. Helens Oregon. He graduated from St. Helens Sr. High School in 1973 and enlisted in the United States Air Force in 1974. He served three years as an aircraft maintenance specialist before cross-training into the computer programming specialty in 1977. Serving as a computer programmer for the Directorate of Logistics, Air Force Data Systems Design Center, and then as NCOIC of Software Support for the Intelligence Data Handling System Branch of the Armed Forces Air Intelligence Design Center, he received both the Air Force Commendation Medal and the Joint Services Commendation Medal.

In 1981 Captain (then Staff Sergeant) Barnes received an Associate in Applied Science in Data Processing from the Community College of the Air Force and was selected to complete his undergraduate studies through the Airmen Education and Commissioning Program. He graduated with "most high honors" from Oregon State University in 1984 with a B.S. in Computer Science and attended Officer Training at Lackland AFB Texas that same year. He received both the Air Force Achievement Medal and Air Force Commendation Medal while serving as Communications and Simulation Software Programmer/Analyst for the PAVE PAWS System Programming Agency, 7th Missile Warning Squadron, from July 1984 through April 1987.

Upon completion of his current graduate studies at the Air Force Institute of Technology, Captain Barnes will assume new responsibilities as an instructor for the Department of Computer Science, Naval Post Graduate School, Monterey, California.

Permanent address: 32525 Highland Rd
Deer Island, Oregon 97054

Bibliography

1. Abbott, R. J. "Program Design by Informal English Descriptions," *Communications of the ACM*, 26, 11: 882–894 (November 1983).
2. Air Force Wright Aeronautical Laboratories. *APEX Users' Guide.* AFWAL, Wright-Patterson AFB, CO., 1987.
3. Alabiso, Bruno. "Transformation of Data Flow Analysis Models to Object-Oriented Design," *OOPSLA '88 Conference Proceedings, ACM SIGPLAN Notices*, 23, 12: 335–353 (September 1988).
4. Alford, Mack. "SREM at the Age of Eight; the Distributed Computing Design System," *IEEE Computer*, 18, 4: 36–46 (April 1985).
5. Andriole, Stephen J. and others. *Storyboarding for C2 Systems Design: A Combat Support System Case Study.* Unpublished paper, George Mason University & International Information Systems, Inc. 802 Woodward Road, Marshall, VA 22115, undated.
6. Balzer, R. and others. "Software Technology in the 1990s: A New Paradigm," *IEEE Computer*, 16, 11: 39–45 (November 1983).
7. Bohm, C. and Jocopini, G. "Flow Diagrams, Turing Machines, and Languages with only Two Formal Rules," *Communications of the ACM*, 9, 5: 336–371 (May 1966).
8. Booch, Grady. *Software Components with Ada.* Menlo Park: The Benjamin/Cummings Publishing Company, Inc., 1987.
9. - - - - -. *Software Engineering with Ada*(Second Edition). Menlo Park: The Benjamin/Cummings Publishing Company, Inc.,1986.
10. Bralick, William A. Jr. *An Examination of the Theoretical Foundations of the Object-Oriented Paradigm.* MS Thesis, AFIT/GCS/MA/88M-01, School of Engineering, Air Force Institute of Technology (AU), Wright Patterson AFB OH, March 1988.
11. Buhr, R. J. A. *System Design with Ada.* Englewood Cliffs: Prentice-Hall Inc., 1984.
12. Byrne, William E. and others. *Structured Hierarchical Ada Representation Using Pictographs (SHARP) Definition, Application, and Automation.* Technical Report Prepared For Electronic Systems Command, Deputy for Development Plans, Hanscom AFB, Massachusetts. Cambridge: Arthur D. Little, Inc. Program Systems Management Co., September 1986.
13. Cox, B. *Object-Oriented Programming: An Evolutionary Approach.* Reading: Addison-Wesley, 1986.

14. Crawford, Bard S. and Jazwinski, Andrew H. "The AdaGRAPH^{TM} Tool for Enhanced Ada Productivity," *IEEE Transactions on Software Engineering*, SE-12, 5: 664–670 (May 1986).

15. Demarco, Tom. *Structured Analysis and System Specification*. Englewood Cliffs: Prentice-Hall Inc., 1978.

16. Diedrech, Jim and Milton, Jack. "An Object-Oriented Design System Shell," *OOPSLA '87 Conference Proceedings, ACM SIGPLAN Notices*, 22, 12: 61–67 (December 1987).

17. Digitalk Inc. *Smalltalk/V Tutorial and Programming Handbook*. Los Angeles: Digitalk Inc., 1986.

18. Department of Defense. *Requirements for the Programming Environment for the Common High Order Language (STONEMAN)*. Washington: Government Printing Office, 1980.

19. EVB Software Engineering,Inc. *An Object Oriented Design Handbook for Ada Software*. Fredrick: EVB Software Engineering, Inc., 1986.

20. Ewing, Juanita J. and Wirfs-Brock, Rebeccca. "Smalltalk isn't Meaningless Chatter," *Computer Design*, 26, 1: 76–79 (January 1987).

21. Freedman, Roy S. "The Common Sense of Object-Oriented Languages," *Computer Design*, 22, 2: 111–118 (February 1983).

22. General Electric Corporation Research and Development Division. *Users' Guide : Interactive Ada Workstation, Prototype Version 1.0*. DOD Contract No. F33615-85-C-1755, General Electric Co., August 1986.

23. Hartrum, Thomas C. and Lamont, Gary B. "Development of a Comprehensive Software Engineering Environment," *Space Operations Automation and Robotics Conference*, Houston (September 1987).

24. Jackson, Michael. *System Development*, Englewood Cliffs: Prentice Hall Inc., 1983.

25. Keen, Peter G. W. "Adaptive Design for Decision Support Systems," *ACM/Database*, 12, 2: 15–25 (Fall 1980).

26. Kelly, John C. "A Comparison of Four Design Methods for Real- Time Systems," *Proceedings of the 9th International Conference on Software Engineering*. 238–251. Washington: Computer Society Press of the IEEE, 1987.

27. Kerth, Norman L. and others. "Summary of Discussions from OOPSLA-87's Methodologies & OOP Workshop," *Addendum to the Proceedings OOPSLA '87, ACM SIGPLAN Notices*, 23, 5: 9–16 (May 1987).

28. Konsynski, Benn and Sprague, Ralph H. Jr. "Future Research Directions in Model Management," *Decision Support Systems*, 2: 103–109 (1986).

29. Korth, Henry F. and Silberschatz, Abraham. *Database System Concepts.* New York: McGraw-Hill, Inc., 1986.

30. Liang, Ting-peng. "User Interface Design for Decision Support Systems: A Self-Adaptive Approach," *Information & Management*, 12: 181–193 (December 1987).

31. Lorensen, W. "Object-Oriented Design," *CRD Software Engineering Guidelines*, General Electric Co., 1986.

32. Magel, Kenneth. "Principles for Software Environments," *ACM SIGSOFT Software Engineering Notes*, 9, 1: 33–35 (January 1984).

33. Nassi, I. and Schneiderman B. "Flowchart Techniques for Structured Programming, " *SIGPLAN Notices ACM*, 8, 8: 12- -26 (August 1983).

34. Novak, Joseph D. and Gowin, D. Bob. *Learning How to Learn.* Cambridge: Cambridge University Press, 1984.

35. Page-Jones, Meilir. *The Practical Guide to Structured Systems Design.* New York: Yourdon Press, 1980.

36. Pascoe, Geoffrey A. "Elements of Object-Oriented Programming," *Byte*, 11, 8: 139–144 (August 1986).

37. Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (Second Edition). New York: McGraw-Hill Book Company, 1987.

38. Riedel, Sharon L. and Pitz, Gordon F. "Utilization-Oriented Evaluation of Decision Support Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-16, 6: 980–006 (November 1986).

39. Ross, Douglas T. "Applications and Extensions of SADT," *IEEE Computer*, 18, 4: 25–34 (April 1985).

40. - - - - -. "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, SE-3, 1: 16–34 (January 1977).

41. Seagle, John P. and Belardo, Salvatore. "The Feature Chart: A Tool for Communicating the Analysis for a Decision Support System," *Information & Management*, 10, 1: 11–19 (January 1986).

42. Seidewitz, Ed and Stark, Mike. "Towards a General Object- Oriented Software Development Methodology," *ACM Ada Letters*, 7, 4: 54–67 (August-September 1987).

43. Simon, H. *The New Science of Management Decision.* New York: Harper & Row, 1960.

44. Sprague, Ralph H. Jr. and Carlson, Eric D. *Building Effective Decision Support Systems.* Englewood Cliffs: Prentice- Hall, Inc., 1982.

45. Stay, J. F. "HIPO and Integrated Program Design," *IBM System Journal*, 15, 2: 143–154 (1976).

46. TRW Defense Systems Group. *Distributed Computing Design System (DCDS) Methodology Guide (Ada Version).* Huntsville: TRW System Development Division, October 1987.

47. Valusek, John R. *The DSS Cube.* Class lecture in OPER 652, Decision Support Systems. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1987.

48. - - - - -. *Concept Mapping.* Class handout distributed in OPER 652, Decision Support Systems. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1987.

49. - - - - -. *The Hook Book.* Class lecture in OPER 652, Decision Support Systems. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, April 1987.

50. Warnier, J.D. *Logical Construction of Systems.* New York: Academic Press, 1975.

51. Webster. *Webster's New Collegiate Dictionary.* Springfield: G. & C. Mirriam Company, 1981.

52. Wegner, Peter. "Dimensions of Object-Based Language Design," *OOPSLA '87 Conference Proceedings, ACM SIGPLAN Notices*, 22, 12: 168–182 (December 1987).

53. Wiener, Richard and Sincover, Richard. *Software Engineering with Modula-2 and Ada*, New York: John Wiley & Sons, Inc., 1984.

54. Wirth, N. "Program Development by Stepwise Refinement," *Communications of the ACM*, 14, 4: 221–227 (April 1971).

Vita

Captain Patrick Denis Barnes [redacted] in 1973 [redacted] enlisted in the United States Air Force in 1974. He served three years as an aircraft maintenance specialist before cross-training into the computer programming specialty in 1977. Serving as a computer programmer for the Directorate of Logistics, Air Force Data Systems Design Center, and then as NCOIC of Software Support for the Intelligence Data Handling System Branch of the Armed Forces Air Intelligence Design Center, he received both the Air Force Commendation Medal and the Joint Services Commendation Medal.

In 1981 Captain (then Staff Sergeant) Barnes received an Associate in Applied Science in Data Processing from the Community College of the Air Force and was selected to complete his undergraduate studies through the Airmen Education and Commissioning Program. He graduated with "most high honors" from Oregon State University in 1984 with a B.S. in Computer Science and attended Officer Training at Lackland AFB Texas that same year. He received both the Air Force Achievement Medal and Air Force Commendation Medal while serving as Communications and Simulation Software Programmer/Analyst for the PAVE PAWS System Programming Agency, 7th Missile Warning Squadron, from July 1984 through April 1987.

Upon completion of his current graduate studies at the Air Force Institute of Technology, Captain Barnes will assume new responsibilities as an instructor for the Department of Computer Science, Naval Post Graduate School, Monterey, California.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/88D-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Strategic Defense Initiative Organization; 8b. OFFICE SYMBOL (If applicable): S/PI	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): The Pentagon, WASHINGTON, DC 20301-7100	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): A DECISION-BASED METHODOLOGY FOR OBJECT-ORIENTED DESIGN

PERSONAL AUTHOR(S): Patrick D. Barnes, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS thesis	FROM _____ TO _____	1988 December	211

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Computer program documentation; Computer systems analysis; Software engineering; Flow charting

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Thomas C. Hartrum
Associate Professor in Electricial Engineering

Approved for release in accordance with AFR 190-1
10 Jan 89

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Thomas C. Hartrum	22b. TELEPHONE (Include Area Code): (513) 255-3576 / 22c. OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

Abstract

The task of object-oriented development raises a new set of design problems. Specifically: how to scope a problem based on objects rather than functions; how to select the best objects; how to encapsulate data structures with the *right* set of operations; and when to stop decomposing a system into objects and begin describing the algorithms that implement those objects' behaviors. The difficulty of making these decisions is increased when the requirements documentation was not developed with an object-oriented paradigm in mind.

Although several software development environments implement an object-oriented design methodology, they seem concerned primarily with "programming in the small" activities, or with providing capabilities for capturing, representing, and storing design decisions once they are made. Recognizing the importance of supporting design decision making, this study focused on the application of decision support system concepts to formulating a methodology for object-oriented design.

This thesis describes an object-oriented design methodology based on the four problems or *decisions* stated above. An object model structure is also defined to provide a foundation for organizing design information. The object model is described by a set of database relations, and includes a three view graphic representation providing block, detail and control flow graphs.

A prototype design tool was implemented to evaluate the methodology. Software for the tool was developed using a PC based implementation of the Smalltalk Object-Oriented Programming Language. Maximum use was made of decision support system techniques such as concept-mapping, storyboarding, the hook book, and adaptive design. As a decision support system, the tool provides the software developer with key requirements specification and software engineering qualitative information to aid in the judgement and design decision making process.