AFIT/GCE/ENG/92-11

EXAMINING A LAYERED APPROACH TO FUNCTION AND DESIGN REPRESENTATION FOR REUSABLE SOFTWARE COMPONENTS

THESIS

Paul Dwight Siebels
Captain, USAF

AFIT/GCE/ENG/92-11

AFIT/GCE/ENG/92-11

EXAMINING A LAYERED APPROACH TO FUNCTION AND DESIGN REPRESENTATION FOR REUSABLE SOFTWARE COMPONENTS

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 Engineering)

Paul Dwight Siebels, B.S.
Captain, USAF

December, 1992

Preface

The purpose of this work was to investigate how software reuse could be improved with particular attention paid to the user interface. Several sources have identified problems directly and indirectly related to how the user was presented reusable software component information. These problems have impacted how successful the reuse of that software would be. This project investigated ways to better present the software to potential re-users.

I began this effort with only a vague idea of where it was headed and even less of an idea of where it would end up. I owe a great deal to the suggestions and direction provided by my faculty advisor, Capt James Cardow and hope the work has helped answer some of the questions that we both had. I especially owe thanks and a great deal more to my loving wife, Constance, who didn't always understand why I was continually in front of the computer but who put up with me anyway. Without her support and acceptance this would have been a great deal harder to complete. Finally I thank the Lord for the patience and strength to complete this at the times when I felt like I didn't know what to do next.

Paul Dwight Siebels

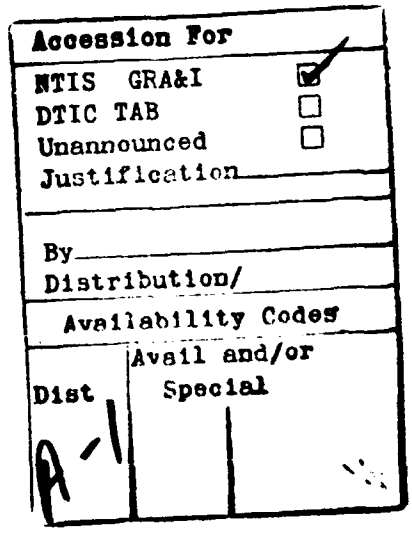

Table of Contents

List of Figures

Abstract

This effort examined ways to improve the effectiveness of reusable software libraries. The main area of investigation was in improving the user interface by finding better ways to present the software components to potential re-users. The first aspect which was considered was in finding an effective representation for reusable software components. A set of criteria was developed for evaluating the effectiveness of software representations. The criteria consisted of generality, expressiveness, understandability, consistency, and resolution. The second aspect which was considered was how to present the software component information to the user to facilitate finding the appropriate component for reuse. A representation framework was examined which advocated presenting reuse information in four layers: component functionality, design information, quality metrics and source code. The first two layers were chosen for further study. Several current representations for software function and design were evaluated using the criteria listed above. These techniques included both textual and graphical approaches. The highest rated representations were then incorporated into a prototype interface for examination by a group of software engineers. Feedback was collected and summarized in a set of recommendations and conclusions.

EXAMINING A LAYERED APPROACH TO FUNCTION AND DESIGN REPRESENTATION FOR REUSABLE SOFTWARE COMPONENTS

I. Introduction

The software crisis has been around for many years, and though many solutions have been investigated, the software crisis seems as much a problem now as it was when it was first described. The solutions have included approaches such as very high level languages, automatic programming, visual programming, object-oriented programming, software reuse, program proving, artificial intelligence and knowledge-based approaches, and many others (6, 15). Each of these approaches has potential for helping reduce the software crisis, but as Brooks asserted, none of them will be a "silver bullet" (6:10). The reusable software library concept is one of the oldest approaches to the software problem. Prieto-Diaz pointed out that McIlroy proposed a software components catalog in 1967 (31:6). Establishing a reusable software library, however, does not by itself solve the problem. Several important aspects of a reusable software library must be addressed, including content, representation, granularity, and indexing (2:9). This thesis examined one aspect of reusable software libraries: representing the reusable software component.

1.1 Background

While software reuse has been recognized for a long time as a desirable activity, it has never fully realized its potential for increasing software productivity and quality

(3:3). One reason for the lack of widespread, successful software reuse is that current reusable software libraries do not adequately present the information necessary for reuse (18:251). A software developer will not reuse software components if the effort to find and adapt the components is more than the effort to create the components from scratch (31:6). The results are that the developer is often frustrated with using the library and dissatisfied with the components found, if any. Thus, the library is not widely used and the promise of software reuse fails to materialize.

1.2 Problem Statement

Current reusable software libraries do not effectively present detailed information associated with reusable software components. Software function and software design specifically are not presented in a way that is beneficial to someone using the library.

1.3 Current Knowledge

Biggerstaff and Richter have identified four activities that reusable software libraries must support: locating components, understanding components, adapting components, and composing components (3:5). Understanding a component is necessary since someone wishing to reuse the component must understand how the component works to use it properly (3:5). This understanding comes in a large part from the representation of the component that the user sees in the library (17:303). Representations for reusable software components can be broken down into three levels: presentation, representation, and implementation (17:303). The presentation of a reusable software component is what the user actually sees. The representation of a reusable software component is how it is logically modeled. The implementation of a reusable software component is how it is physically retained. (17:303) The first two levels of representation were the focus of this thesis.

One issue for reusable software libraries is deciding what should be included in the library as a reusable component. Arango and Prieto-Diaz have pointed out that it is difficult to identify what information will make the best type of reusable component (2:9). Frakes and Gandel, among others, have stated that any product developed during the software lifecycle is a candidate for reuse. This would include such objects as operational concepts, functional requirements, design information, source code, testing products, maintenance products, and user documentation. (17:302) Caldiera and Basili contend that all these objects should be reused together. More than just the code must be reused since all the objects are related and need to be defined in their entire context. (9:61) Design particularly is an area that has much potential for reuse beyond the design of the software component itself. Several articles have stressed that many parts of the design *process* should be retained (1, 14). Conklin states that information such as design decisions, alternatives, assumptions, and rationale are important for understanding the software system (14:533).

Given so many different types of reusable software components, the question is how are the components presented to someone using a software library. One approach that has been proposed is to use hypertext or hypermedia systems to integrate the various types of reusable component information and maintain relationships between them (3, 36). Another approach is to arrange the component information in tiers (10). The components would be presented in four layers: component functionality, design information, quality attributes, and source code (10:10-12). One of the concepts behind the layered presentation is showing the component information at various levels of detail for many similar components to allow comparison between the components. This is similar to the way common electronic hardware is presented in a Transistor-Transistor Logic (TTL) data book. (11) This layered approach fosters reuse by providing proven, specific components to be combined into larger, more general functions in the same way common TTL components are combined. This

approach has the advantage of retaining the context for the components as advocated by Caldiera and Basili.

Two of the four layers mentioned above were the subject of this work: software function and software design. Software function and design representations fall in two general categories, textual and graphical methods. Textual descriptions of software function were developed along several different approaches. Some example approaches include:

1. Text Descriptions (19)
2. Keywords and Facets (7, 31)
3. Frames and Forms (15, 27)
4. Formal and Logical Approaches (20)

The graphical approaches to describing software function that have been proposed include:

1. Data Flow Diagrams (37)
2. Semantic Nets and Entity Relationship Diagrams (16)

Software component design has been presented in many cases by extensions to the techniques used for software function presentation. One additional textual method is program design language (32). Other graphical methods include:

1. Structure Charts (37)
2. Plan Calculus (34)
3. Schemas (26)

Finally, some techniques are at a higher level of abstraction or take on a wider view of representation. Two such techniques are:

1. Hypertext (13)
2. A Metamodel (23)

These techniques will be detailed in Chapter II.

1.4 Research Questions

Several questions exist in the area of software reuse and specifically in the area of software component representation. For the two layers discussed above, software functional representation and design representation, the following questions will be addressed by this thesis.

1. How is software functionality currently represented?
2. How can software functionality be presented to a user?
3. What methods are effective in representing reusable software component design and related design information?
4. How can software designs and design information be presented to a user?

1.5 Methodology Overview

The first step in the process to answer the research questions was to set up the evaluation criteria that would be used to compare various software function and design representations. As Karat emphasized, the evaluation criteria or objectives are a necessary first step since without them the evaluation process would be very difficult and possibly useless (24:892). The next step was to examine the current software function and design representations and record how well the representations met the various criteria. The third step was to choose the two or three representations that were the closest match to the evaluation criteria and incorporate them into a prototype system. The next step was to produce a set of questions concerning the prototype representations to determine how effective the representations were. These questions were answered by a group of users after working with the prototype. The advantages of using a questionnaire were ease of use, low cost, applicability, and suitability. The questionnaire was applicable and suitable since it could be used any time clear questions could be formed, and it has been widely used in experimental

work (24:894,896). Finally, after the questionnaire was completed, the answers were analyzed to determine if the representations were effective or if some changes in using the representations for a reusable software library could be suggested.

1.6 Thesis Overview

This chapter has introduced the issue of software component representation for reusable software libraries. Chapter II will continue with a more detailed look at how software function and software design are represented. Various representation techniques will be examined with their associated benefits and limitations. Chapter III will discuss the methods that were followed in developing the prototype. Chapter IV will present the results of the work on developing the prototype. Chapter V summarizes with some conclusions that were drawn from the work and recommendations for further investigation.

II. Literature Review

Representing reusable software is one of the difficulties that has contributed to the low percentage of software reuse (17:302). The following sections of this chapter present examples of the work that has been published that addresses how software is represented. The first section is about representing software function. The second section is about representing software design. The software function and design representations fall in two general categories, textual and graphical methods. These first two sections include discussions on the textual and graphical methods available. The third section covers two representation techniques, hypertext and a metamodel, which present a wider view than the typical representations.

2.1 Representing Software Function

Representing the function or purpose of software has been a task that has been around as long as software itself. Originally, the representation may have been just a short description written by the programmer to remember what the code did. Then subroutine libraries appeared and usually included some written documentation describing what the subroutines did. Now, reusable software libraries have been developed. Representing the function of the reusable software component is a key factor in the effectiveness of the library (19:147). The following subsections on textual and graphical methods describe various approaches to representing the function of software. In these discussions, software requirements are considered to specify software function.

2.1.1 Textual Methods Textual descriptions of software function were developed along several different approaches. Four example approaches are discussed next describing some work that was done using the various formats.

2.1.1.1 Text Descriptions Simple text descriptions always have been used to describe the function of software. Comments in the source code were one of the first methods programmers learned in introductory programming classes on how to describe software. The comments described the purpose of the overall code and the various parts of the code. The comments, though, were usually informal and often inconsistent. Some work has been done in making comments more effective. Frakes and Nejmeh proposed standardized headers for software components to present a description of the function of the component as well as other useful information (19:147). The types of information recommended for the header were the name of the component, author, creation date, a short abstract, a longer description, keywords, size, complexity, performance, inspection data, testing data, and much more. Part of the reasoning for such a standardized header was to help ensure the quality of the components in a software library. (19:148) The header was then used as a basis for the indexing scheme of a library system by automatically extracting important words from the header for indexing use. One problem the authors pointed out with their approach was that the syntactic and semantic information about relationships between the key words was not retained. (17:308)

2.1.1.2 Keywords and Facets A similar approach to a text description involved using one or more keywords to define the software component. This approach was particularly useful as part of the search algorithm for the software library. One library system that used this approach was the Reusable Software Library (RSL) (7). RSL used specially labeled comments to document particular attributes of the software component. One of the attributes was a hierarchical category code that described the functionality of the component. This category code was used as part of the classification strategy of the library. In addition to the category code, another attribute defined by RSL consisted of up to five descriptive keywords to further

define the functionality of the component. These keywords were also used in the classification strategy but were not in any hierarchy. (7:132)

Another keyword approach was advocated by Prieto-Diaz. This approach to defining a software component used specific keywords to define several facets of the software component. Each facet was an aspect or dimension of a particular domain (31:8). In the software domain, component functionality was represented by the function, object, and medium facets. The function facet was the action that the component performed, the object facet was the receiver of the action, and the medium facet was the location of the action. System type, functional area, and setting were three additional facets that defined the environment that the software component was designed to operate in. (31:10) Prieto-Diaz reported that a library system using the faceted approach was used successfully at GTE Data Services. Initial indications showed a 14% reuse factor with an estimated savings of $1.5 million. (30:94) Another library using the faceted approach was also started with similar high expectations (30:96). Figure 1 shows the six facets proposed by Prieto-Diaz with a partial listing of keywords for each facet.

The keyword and faceted approaches were not a perfect answer to the representation issue. One problem was that the approaches depend on finding one or more keywords that distinctly defined the function. When the keywords were found, they still had the drawback pointed out by Devanbu and others that the semantics of the keywords were not available for searching or retrieving the software components. The search and retrieval algorithms could not "in any way infer the 'meaning' of the special set of keywords used in the query" (15:38). The faceted approach was more accommodating for the usual search, modify, repeat retrieval cycle than the simple keyword approach, but it was weak in that it could not represent any constraints between the keywords (15:38).

Function	Object	Medium	System	Area	Setting
add	arguments	array	assembler	accts payable	advertising
append	arrays	buffer	compiler	auditing	association
close	blanks	cards	DB mgmt	billing	auto repair
compare	buffers	disk	evaluator	bookkeeping	barbershop
create	characters	file	interpreter	budgeting	car dealer
decode	digits	line	line editor	CAD	cemetary
deleted	files	list	network	cost control	circulation
divide	functions	printer	retreiver	DB analysis	cleaning
expand	integers	stack	scheduler	DB design	clothing
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Figure 1. Example List of Facets (31:10)

2.1.1.3 Frames and Forms Frames were an object-based approach to representing software. "Frames are data structures, composed of slots and fillers, used for knowledge representation" (17:310). The frame itself represented a class of objects, the slots were the attributes of the objects, and the fillers were used to define a specific object (15:40). Devanbu and others used a frame-based approach called LaSSIE to implement a library system that presented functional, architectural, feature, and code views of the components in the system (15). The frame based approach afforded two advantages over other approaches. The first advantage was that it was good at describing complex relations between objects that had a hierarchical structure. The second advantage was that constraints on or between attributes of the object were expressed by putting limitations on the fillers. (15:40) A simple example frame is shown in Figure 2.

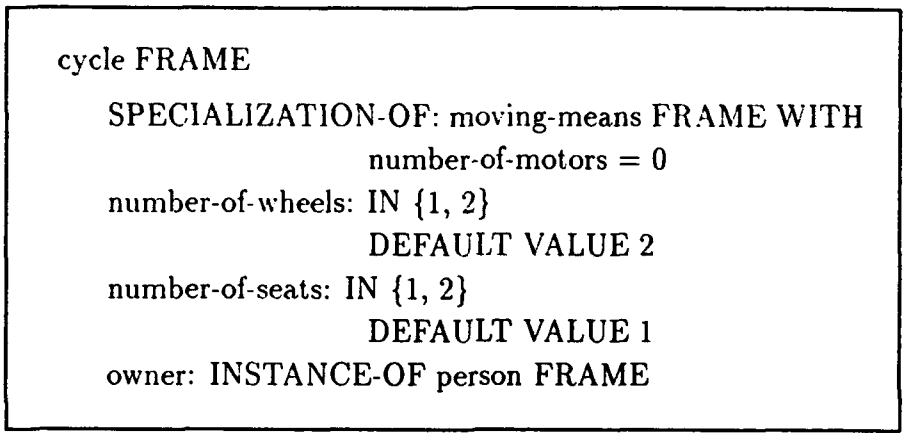

Figure 2. Example Frame Representation (16:1432)

Matsumoto reported on work he had done on representing reusable software using forms (27). The forms presented the software at four levels of abstraction. These levels were requirements, design, program, and source code. (27:160) The requirements level specified the relationships and constraints between external objects and the software. The design level included the data structures, data flows, functions, and control flows. The program level presented the external design of the software modules, and the source code level presented the internal design of the modules. (27:160,164,167) The form at each level consisted of a structured description of the information at that level using an Ada-like design language. The forms included descriptions of elements that could be customized for different applications with acceptable ranges of values for the elements. Finally, the forms included traces between the requirements, design, programs, and source code. (27:184) Figure 3 shows examples of the structure of the first three levels of forms.

2.1.1.4 Formal and Logical Approaches In the formal or logical approaches, information was represented by formulas and logical expressions. One example of this type of approach was a language called Ted described by Franke (20). Ted was a formal language that was designed specifically to be able to describe

Form(1) - - requirement level:
object type: EXTERNAL_ENTITY, INTERNAL_ENTITY, PROCESS, INPUT_INTERFACE, OUTPUT_INTERFACE, DATA, DATA-SET, EVENT;
relationship type: with, trigger/triggered_by, use/used_by, comprise/comprised_in, acknowledge/acknowledged_by;

Form(2) - - design level:
object type: FUNCTION, INTERFACE_FUNCTION, DATA, FILE, SIGNAL;
relationship type: with, activate/activated_by, call/called_by, converse/conversed_by;

Form(3) - - program level:
object type: PACKAGE, SUBPROGRAM, TASK, PROCEDURE, FUNCTION, DATA;
relationship type: with, invoke/invoked_by, call/called_by, cause/accept;

Figure 3. Example Forms (27:172-173)

the purpose of a software component. It communicated the purpose by describing actions that were prevented, enforced, or conditionally introduced (20:41). The advantage to using a formal approach was that it had well defined semantics and had available an established set of inference rules (34:320). A problem with the formal approach was that it could be difficult to comprehend the structure of large numbers of formal or logical expressions (16:1432). An example of a representation using the formal language Z (22) is shown in Figure 4.

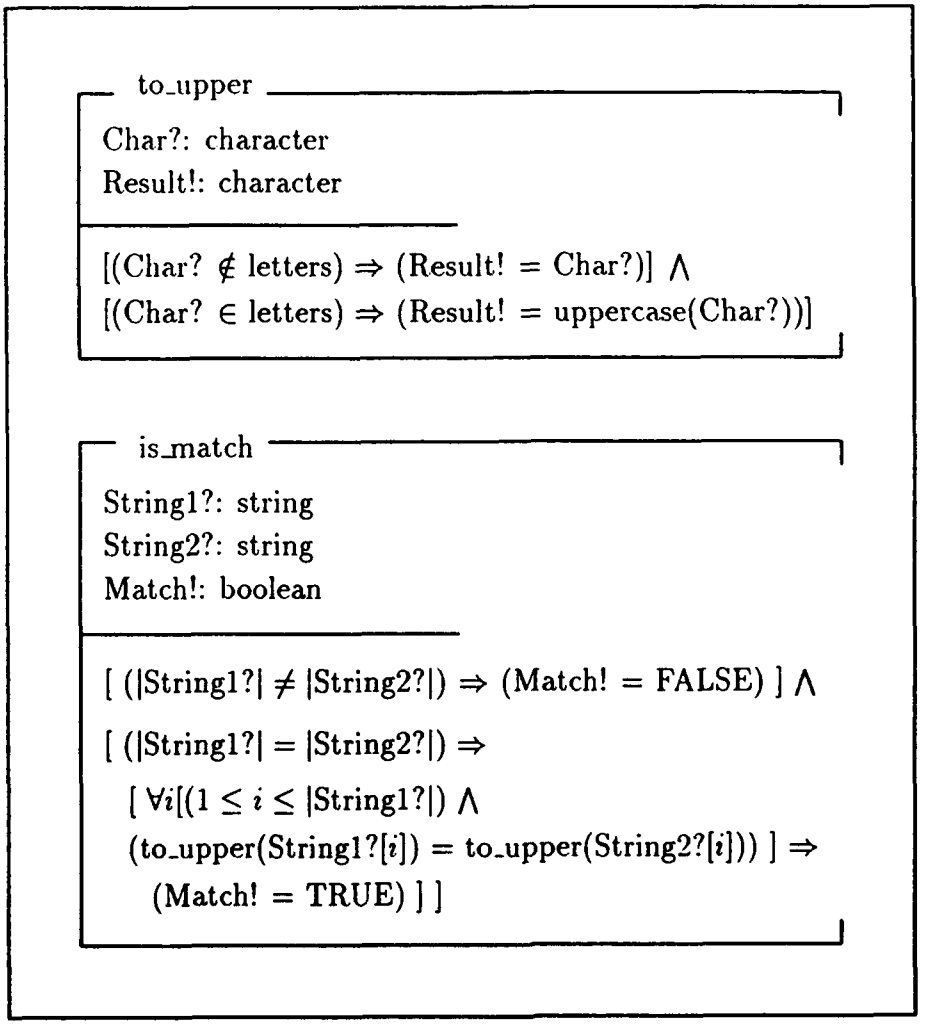

Figure 4. Example Formal Representation

2.1.2 Graphical Methods Graphical methods were also developed to represent the function of software components. While the earliest graphical representations were originally created on paper, the graphics capabilities of computers have enabled tools to be developed to automate much the work. Some examples of the graphical methods are presented next.

2.1.2.1 Data Flow Diagrams Data flow diagrams (DFDs) were used as part of the technique of Structured Analysis (SA) as described by Yourdon (37). DFDs fell into the category of "classical design techniques," with the similar approaches of Hierarchical Input Output (HIPO), Warnier-Orr and Jackson Development Methods (35:10). The DFDs showed how data moved into and out of the system components and what processing was done on the data. DFDs and the other classical methods were good at encouraging communication between people and enhancing understanding of the system components themselves (35:10). The problem with using SA or any other classical method as a representation for software function was that it was not effective since

> it does not have a semantic basis for clearly denoting or explaining, either to man or machine, the meaning of ideas and concepts represented within it or to mechanically communicate the rationale behind the choices and intended interpretation of a diagram. (35:23)

An example DFD is shown in Figure 5.

2.1.2.2 Semantic Net and E-R Diagrams Semantic nets and entity-relationship (E-R) diagrams both used a graph-based approach to software functional representation where the nodes of the graph represented objects or concepts and the links corresponded to relationships between the nodes (16:1432). The advantages to a semantic net representation included being easy to represent information (17:309) and providing easy traversal of the information (16:1432). A problem with

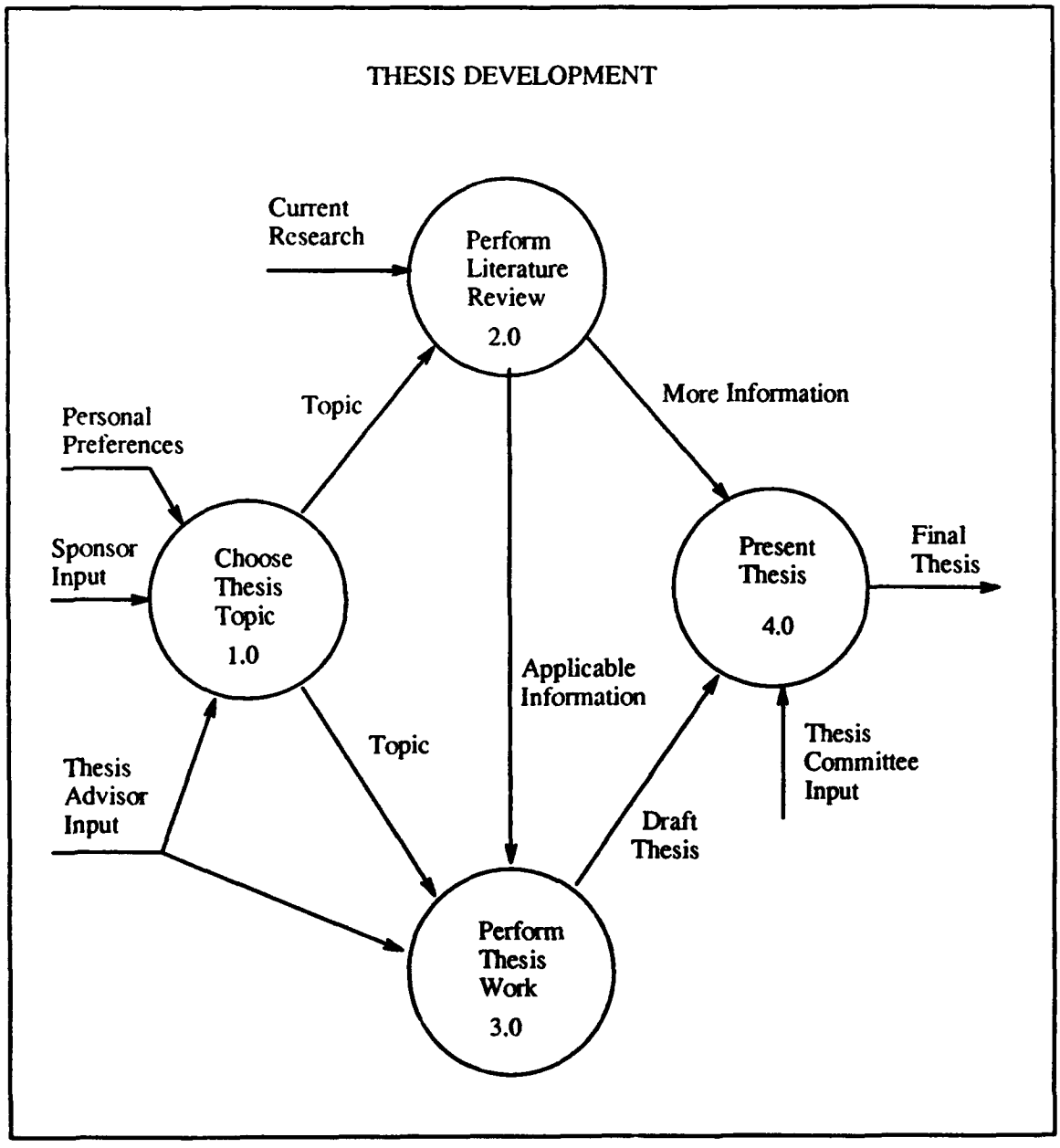

Figure 5. Example Data Flow Diagram (DFD)

semantic nets was that there were no common semantics among various implementations (16:1432). This contributed to the difficulty of not being able to do any reasoning about the information represented (17:309). Dubois and others proposed using a combination of semantic nets and logical expressions for use in requirements representation (16:1436). The goal of this approach was to combine the advantages and overcome the disadvantages of both methods (16:1436). This representation adopted a world-oriented requirements view. This view included the expected behavior of the software component, the interaction requirements between the software and its environment, and the requirements or assumptions about the environment itself. (16:1431) An example of a simple semantic net is shown in Figure 6.

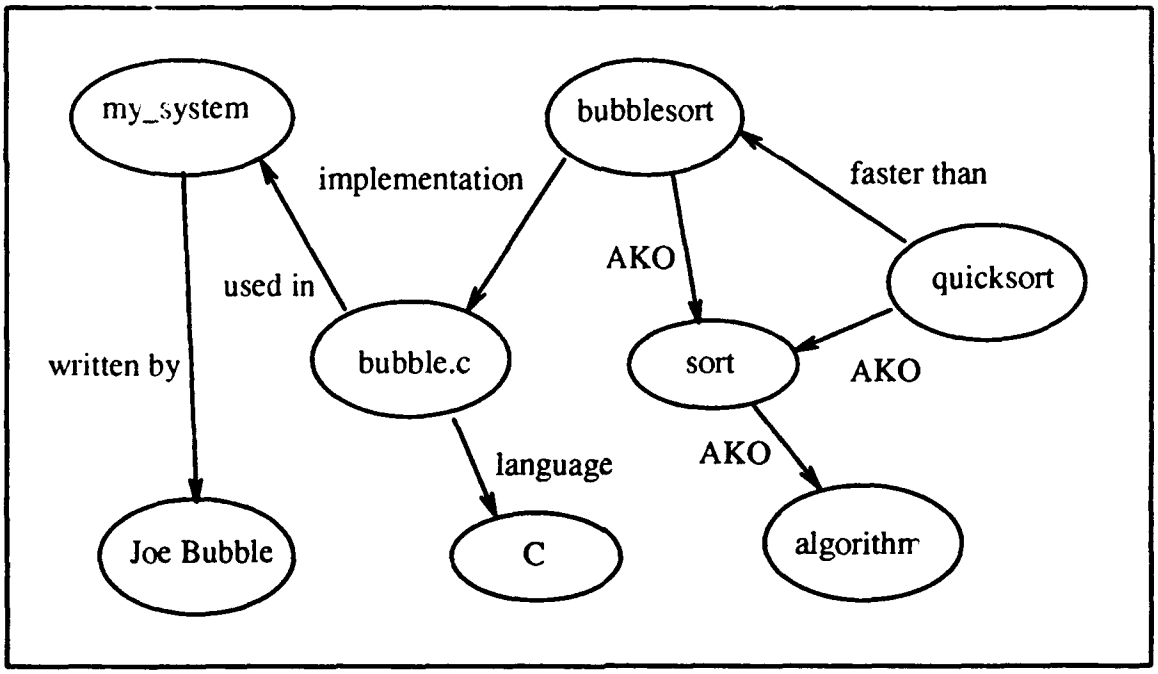

Figure 6. Example Semantic Network (17:309)

2.2 Representing Software Design

The representations for software design in many cases were the same as the representations for software function only extended to a more detailed level. Some approaches that were used for both software function and design representation included text descriptions, frames and forms, formal and logical expressions, data flow diagrams, and semantic nets. Beyond these methods, some representations were more oriented to just software design. The following sections on textual and graphical methods describe these additional approaches.

2.2.1 Textual Methods One of the main textual representations of design beyond those already discussed was program design language (PDL), also known as structured English. A PDL was simply a description of the design using natural language text embedded in structures such as loops or conditional statements. The use of Ada as a design language has been promoted for several years (32). An example of using a PDL was discussed earlier in the description of the work with forms done by Matsumoto. In the forms, Ada was used as the design language because the implementation language was going to be Ada and using a different design language would have added additional cost without adding any benefit (27:184). Some advantages advocated for a PDL were that they were easier to create, revise, and understand for a human's perspective (8:485). A generic PDL example is shown in Figure 7.

2.2.2 Graphical Methods Several additional graphical techniques were available for representing software designs. Some techniques have been around for several years, while others were recent work. The following three methods are examples of the work in this area.

```
SORT (Table, Size_of_Table)

if Size_of_Table > 1
  do until no items were interchanged
    do for each pair of items in table (1-2, 2-3, etc)
      if first item of pair > second item of pair
        interchange the two items
      end if
    end do
  end do
end if
```

Figure 7. Example of Program Design Language (PDL) (8:486)

2.2.2.1 Structure Charts Structure charts were a graphical design representation technique that presented software as a hierarchical organization of lower level components (37:417). The structure chart, as its name implies, showed the structure of the software, including the data flow, control flow, repetition, and conditional module invocation. The design goals for using structure charts were high quality, error free software. These goals were accomplished by following several design guidelines, such as maximizing intramodule cohesion, minimizing intermodule coupling, and maintaining a reasonable module size. (37:421-422) An example structure chart is shown in Figure 8.

2.2.2.2 Plan Calculus Rich and Waters reported on a representation technique called Plan Calculus that represented reusable components as plans (34). Each plan had three parts: the algorithmic portion represented by plan diagrams, the non-algorithmic aspects captured by logical annotations, and program transformations documented by overlays (34:323-325). The plan diagrams showed the computations, data flows, and control flows using hierarchical data flow schemas (34:323).

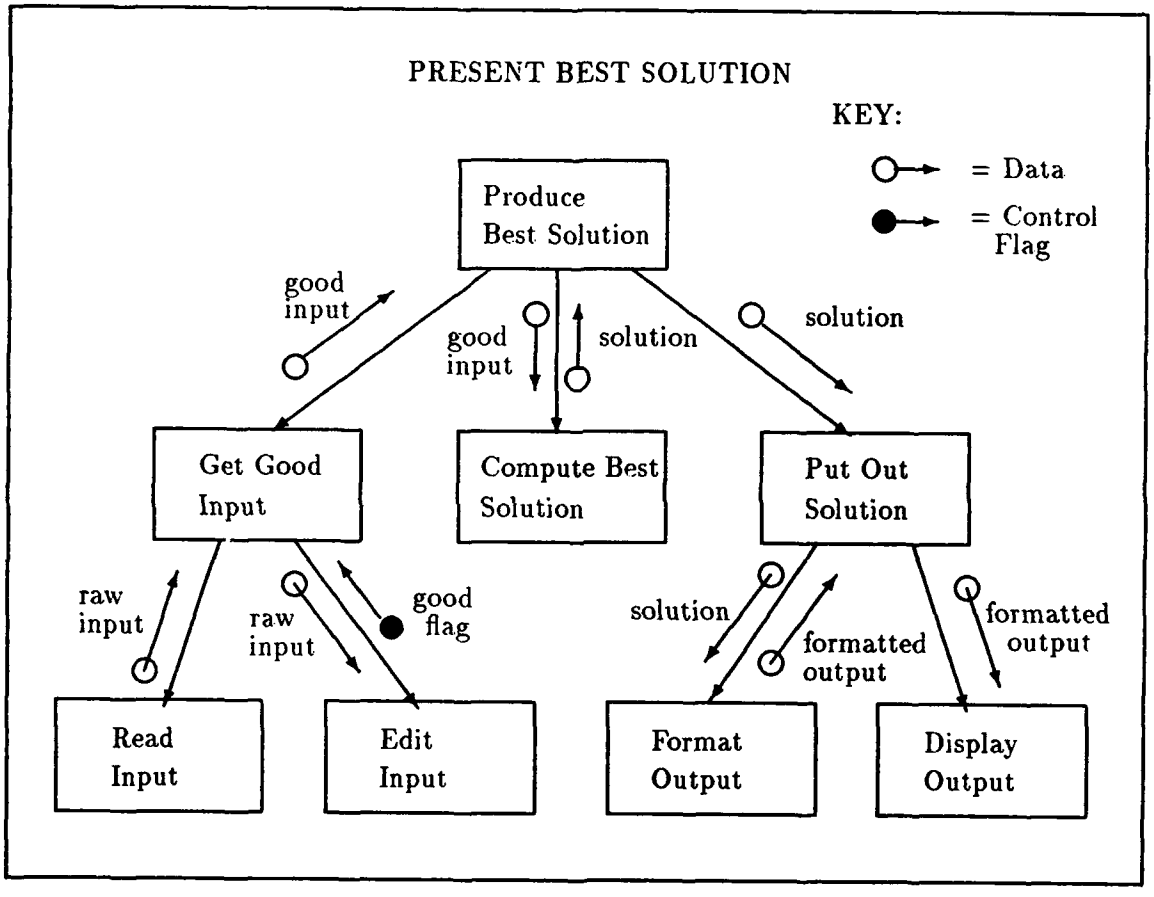

Figure 8. Example of Structure Chart (21:130)

The annotations on the plan diagrams captured the non-algorithmic aspects of a component, such as preconditions, postconditions, constraints, and dependencies. These aspects were represented by predicate calculus assertions. (34:324) The transformations, documented by overlays on the plan diagrams, simply were mappings between corresponding roles or portions of two similar plans (34:325). The advantages expected for using Plan Calculus as a component representation included high expressiveness, language independence, semantic soundness, and machine processability (34:327). An example of a plan calculus is shown in Figure 9.

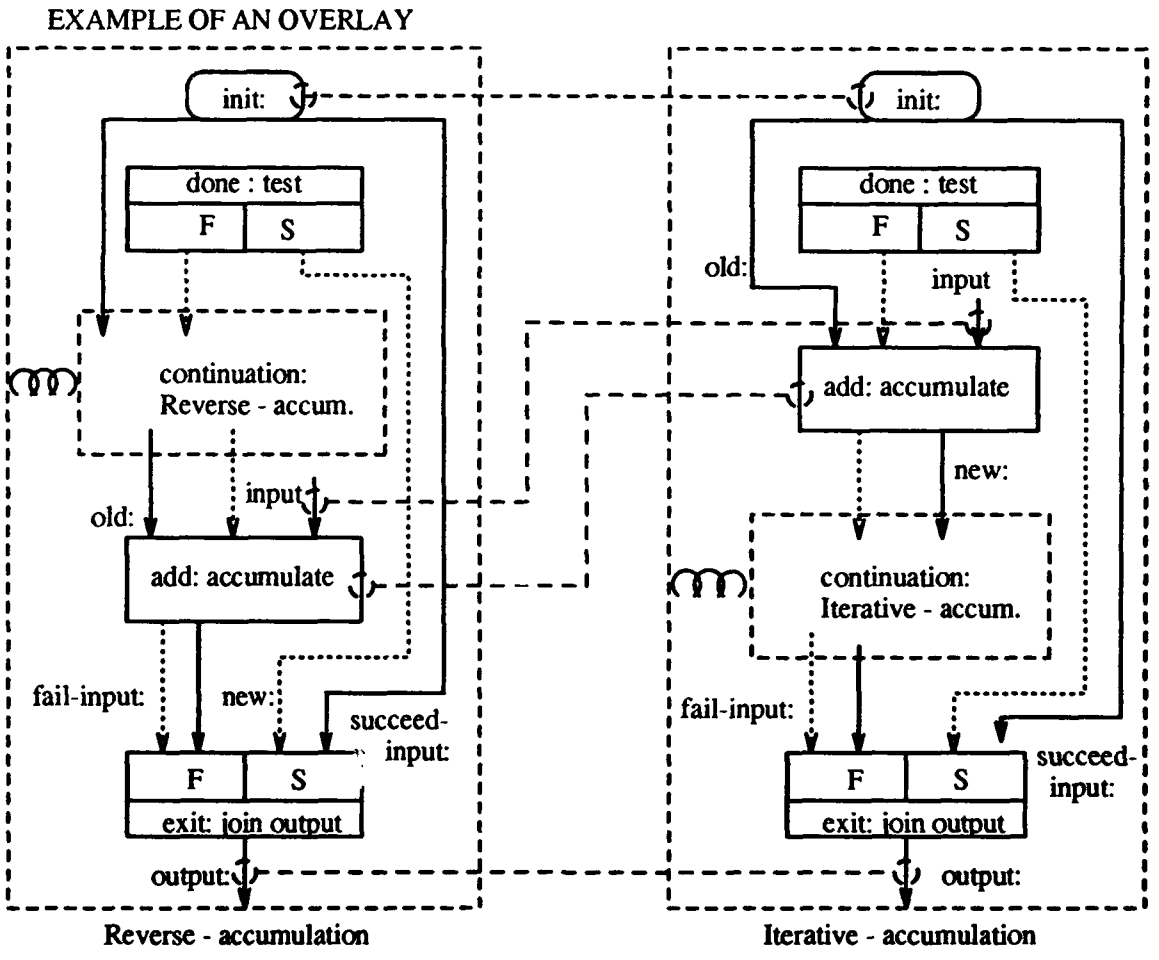

Figure 9. Example Plan Calculus (34:326)

2.2.2.3 Schemas Lubars and Harandi reported on IDeA (Intelligent Design Aid), a knowledge-based design environment that used design schemas to represent reusable designs (26:346). The design schemas represented design solutions that were abstracted for a particular domain (25:163). The design schemas themselves were represented using data flow diagrams, though the concepts captured in IDeA were independent of any representation method (26:348). The goal of IDeA was to improve the effectiveness of reuse and reduce the number of errors in the software by reusing good designs early in the development process (26:346). A potential pitfall was the high reliance on a knowledgeable domain expert to perform the initial domain analysis used to populate IDeA's knowledge base (25:177). The example schema shown in Figure 10 shows how an abstract design could be instantiated in two different reusable functions.

2.3 A Wider View

The following two representation techniques took a broader approach than the previous techniques. Hypertext dealt with how various pieces of information for a component could be arranged to provide a better representation, regardless of the format the pieces of information were in. The metamodel proposed merging several different representation techniques into one model, so that a complete representation of the component could be maintained. These techniques are described in more detail next.

2.3.1 Hypertext Hypertext was a method of allowing non-sequential access to sections of text via machine processable links (13:18). Examples of hypertext systems included IBIS, NoteCards, and PlaneText (13:21). The advantages of using a hypertext approach included allowing related information to be readily accessible in any order and allowing information to be structured in the most effective manner

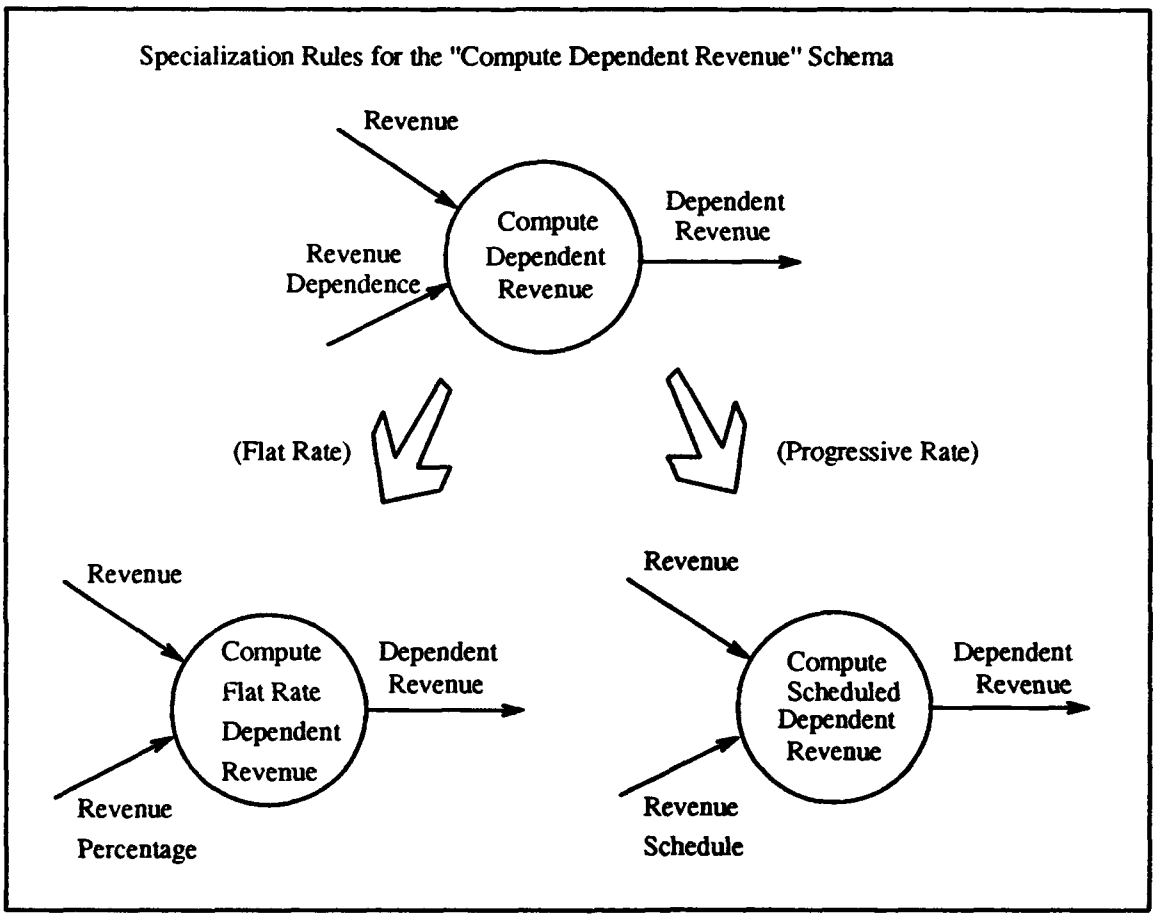

Figure 10. Example Schema Diagram (26:358)

(13:38). This made hypertext an excellent representation method for people (35:14). With today's high power computers and corporate workstations, graphics capabilities could be added to the hypertext system to create a hypermedia implementation. In using hypertext or hypermedia for representing a component, several types of information could be combined and presented in the format that was best for that part of the representation. A drawback to a hypertext approach was getting "lost in space", or losing orientation in all the data connections (13:38). This was usually overcome by providing some type of overview picture or map to the user (13:19). Figure 11 shows how a hypertext system works.

2.3.2 A Metamodel Jordan and Davis proposed a metamodel combining several approaches to representing the requirements for software objects (23). The different views or approaches that were combined in the metamodel include data flow diagrams (DFDs), object oriented analysis (OOA), and a finite state machine (FSM). The reasoning for this approach was that no single technique could adequately represent all aspects of the object. (23:472) By combining the three approaches, the strengths of each were captured. The DFDs provided data and control flow, OOA provided descriptions of objects, attributes and operations, and the FSM provided the system behavior through transitions in the state of the system (23:471). Using such a model provided an independence from design methodology, a consistent view across differing methodologies, and a more complete view of the requirements for the software object (23:478). An example metamodel showing combined DFD, OOA, and FSM views is shown in Figure 12. While the figure shows all three views at once, the metamodel is normally an internal model with only one view extracted from it at a time.

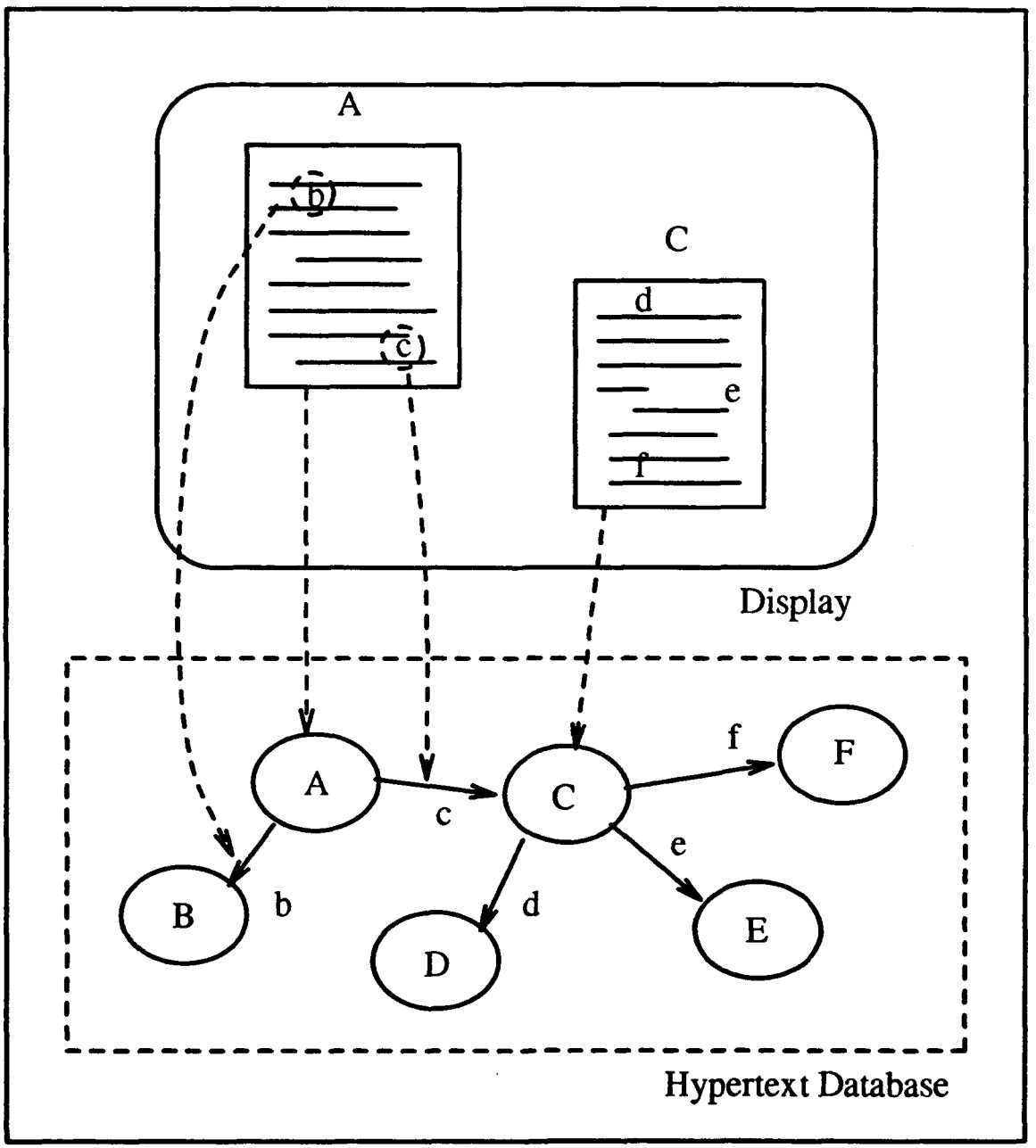

Figure 11. Example Hypertext System (13:18)

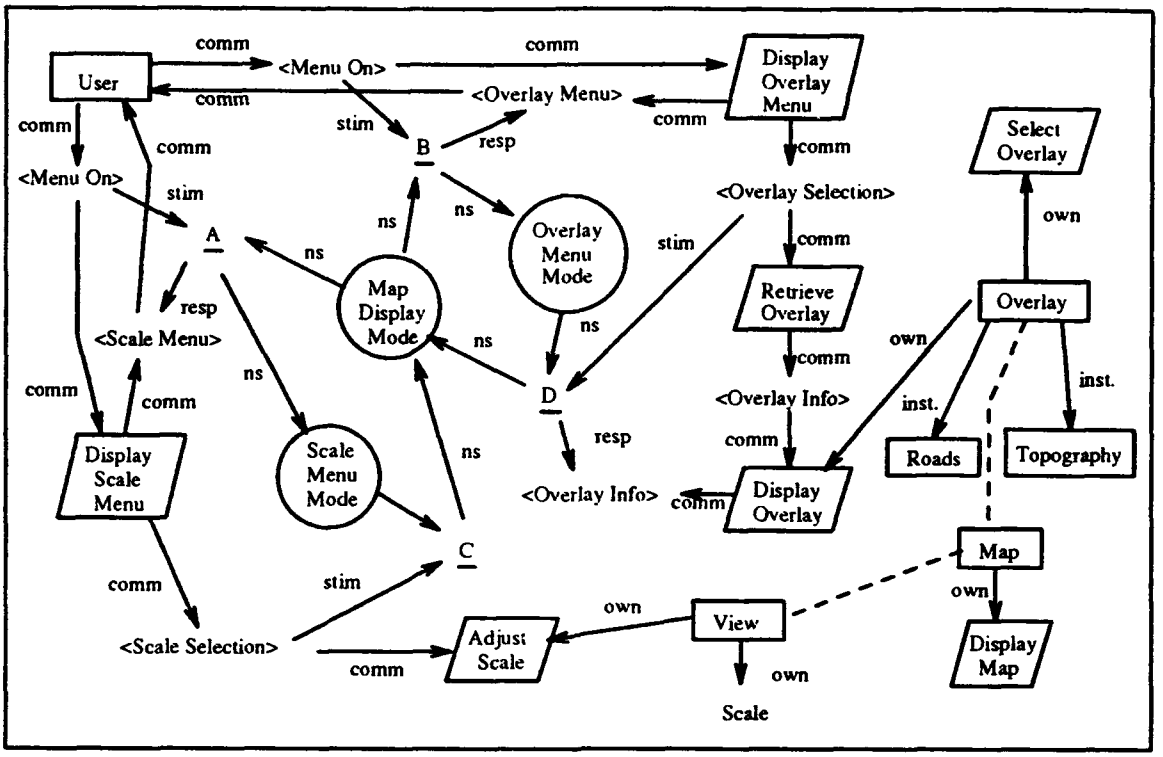

Figure 12. Example Meta-model (23:477)

2.4 Summary

The previous sections discussed the various techniques for representing software function and software design. The advantages and disadvantages of the different techniques were presented for the techniques that were used for actual software development or software reuse systems. The desired goals of newly proposed systems were also described. All the methods, both textual and graphical, had some strengths and weaknesses. Finally, two techniques that presented a wider view were examined. These techniques recognized that no one technique could completely represent software components, and so they presented different approaches to providing more information using several techniques.

III. Methodology

As introduced in Section 1.4, the purpose of this effort was to answer the following research questions:

1. How is software functionality currently represented?
2. How can software functionality be presented to a user?
3. What methods are effective in representing reusable software component design and related design information?
4. How can software designs and design information be presented to a user?

A six step process was outlined in Section 1.5 to answer the research questions. The six steps were:

1. Establish evaluation criteria for representations.
2. Evaluate current function and design representations.
3. Develop a prototype using highest evaluated representation(s).
4. Develop a questionnaire about prototype.
5. Evaluate prototype using questionnaire.
6. Analyze questionnaire results.

Each step is presented next in more detail.

3.1 Evaluation Criteria

This step was broken down into two substeps. The first was to determine what characteristics of a representation were important and then define a set of criteria based on those characteristics. The second substep was to develop a scale and a set of guidelines for rating the various representations using the criteria. These substeps are discussed next.

3.1.1 Selecting Specific Criteria In order for the software library to be used, the components must be understandable. Understandability is a primary issue of the representation (17:303). To enhance understandability, several authors have suggested characteristics that a representation method or the reusable component itself should possess. Dubois proposed the following characteristics to improve the communication effectiveness of knowledge representations: formality, deductive power, abstraction capability, and conversion to other languages (16:1435-1436). Frakes proposed several issues to consider for selecting a reuse representation, including consistency, expressiveness, comprehensibility, presentation, library operations, administrative issues, and implementation issues (17:303-304). Matsumoto suggested the following attributes for reusable components: generality, definiteness, transferability, and retrievability (27:159). Rich identified the following properties of an effective representation: expressiveness, combinality, semantic soundness, machine manipulability, and language independence (34:315). Finally, Webster recommended some requirements for a representation consisting of: rich representational features, machine processability, inferencing mechanisms and a powerful user interface (35:7-8). Since this thesis was concerned with evaluating several representation methods, a set of evaluation criteria was developed that incorporated the characteristics suggested above. The definitions of these evaluation criteria and the rationale for their selection are presented below:

1. *Generality*: This evaluation criterion indicates whether the representation was independent of programming languages, development methodologies or specific applications (17, 27, 34). The reason for this criterion was:

    - The representation should not be dependent on any one computer language or specific application since it will reduce the effectiveness of representing components of other languages or applications (34:315).
    - The representation should not be dependent on the development methodology so that developers unfamiliar with the methodology can understand the components (27:159).

2. *Expressiveness*: This criterion indicates how well the representation technique could represent different types of components for the various layers of information (17, 34, 35). The reason for this criterion was:

    - The representation should be able to represent many kinds of reusable components to be of value (34:315).

3. *Understandability*: This criterion indicated whether the representation could be easily comprehended by people (17). The reason for this criterion was:

    - Understandability is critical for a representation since a component must be understood before it will be reused (17:303).

    - Someone wishing to reuse the component must understand how the component works to be able to use it properly (3:5).

4. *Consistency*: This criterion shows whether there was agreed upon notation and rules for interpreting the representation (16, 17). The reason for this criterion was:

    - Consistency enables common interpretation in the transfer of large amounts of information between people over time (16:1435).

    - Consistency is a part of the software quality factor of correctness as defined in (5:3-12), which was rated high to very high on the level of importance to Air Force applications in (28:3-8).

5. *Resolution*: This criterion indicated how well the representation can represent small differences between similar components (11). The reason for this criterion was:

    - Having a way to choose among similar components is critical for the user (7:133).

Three of the criteria above were composites of some of the desirable software representation characteristics that were mentioned previously. The criterion *generality* incorporated the characteristics: scope, generality, transferability, and language independence. The criterion *understandability* embodied comprehensibility and conversion to other languages. Finally, the criterion *consistency* combined the characteristics: formality, consistency, semantic soundness, and deductive capability. The other two criteria simply reflected the two characteristics expressiveness and definiteness.

The representation characteristic "machine processability" suggested by many authors was not specifically included because it was implicity related to consistency. When a representation is standardized or widely used, which for this effort is defined as having high consistency, software tool developers quickly include it in automated software development tools to take advantage of its widespread use. Thus, a representation that is rated high in consistency will either have or soon have the characteristic of machine processability. Other characteristics were not included because they were already implicitly contained in the concept of the layered approach. These include abstraction capability, presentation, combinality, powerful user interface, retrievability, and other library operations. These characteristics are the basis of how the layered approach is implemented. Two of the issues raised by Frakes were important enough that they were addressed separately. They were administrative issues (the financial cost of creating and updating the reusable library) and implementation issues (the computer resource cost for using the library). These issues are addressed in Section 3.3.1.

3.1.2 Developing Evaluation Guidelines Once the evaluation criteria for the representations were chosen, a scale for each criterion needed to be defined. A forced choice scale consisting of the five choices: Very Low, Low, Nominal, High and Very

High was used following the manner of (29). To help reduce the subjectivity of the scale, the following guidelines were developed to help in evaluating the specified characteristics of the representations.

1. Guidelines for evaluating representation *generality*

   - *Very High*: no language, methodology or application constructs or terms were part of the representation
   - *High*: minimal use of common language, methodology or application constructs or terms, where common constructs and terms were those used by three or more different languages, methodologies or applications
   - *Nominal*: moderate use of common and minimal use of specific language, methodology or application constructs or terms, where specific constructs and terms were those used by only one or two languages, methodologies or applications
   - *Low*: general use of common constructs and terms or moderate use of specific language, methodology or application constructs or terms
   - *Very Low*: general use of specific language, methodology or application constructs or terms.

   [Note: See Figure 13 for examples of common and specific language, methodology and application constructs and terms.]

2. Guidelines for evaluating representation *expressiveness*

   - *Very High*: there were basic representational elements that correspond with almost all component elements at a given layer
   - *High*: there were basic representational elements that corresponded with some component elements, and other component elements could be represented by combinations of representational elements
   - *Nominal*: most component elements could be represented by simple combinations of representation elements and others by complex combinations

Specific language constructs and terms	Common language constructs and terms
task (Ada)	if... then... else
perform (cobol)	function; call
common (FORTRAN)	do; loop
set (Pascal)	integer; real
Specific methodology constructs and terms	**Common methodology constructs and terms**
coupling, cohesion (SD)	requirements analysis
objects, attributes (OOD)	abstraction
model, modelling (JSD)	module; modularity
Specific application constructs and terms	**Common application constructs and terms**
MAC Standard	protocol
X400	handler
DDN	file transfer

Figure 13. Examples of Language, Methodology and Application Constructs and Terms

- *Low*: most component elements required complex combinations of representational elements to represent

- *Very Low*: most component elements could not be usefully represented

[Note: see Figure 14 for a list of component types and component elements for each layer.]

3. Guidelines for evaluating representation *understandability*

   - *Very High*: only general knowledge in software development required

   - *High*: some general knowledge of representations required

   - *Nominal*: some training or experience with representations required

   - *Low*: some training or experience in the specific representation required

   - *Very Low*: extensive training and experience in the specific representation required

4. Guidelines for evaluating representation *consistency*

   - *Very High*: an official standard or established set of rules for notation and interpretation existed

   - *High*: a widely accepted notation and set of rules for interpretation existed with minor variations (a de facto standard)

   - *Nominal*: a commonly accepted notation and set of rules for interpretation existed with moderate variations, which included non-standard extensions to accepted or standard notation

   - *Low*: only a few major variations of commonly accepted notation and rules for interpretation; or research work not widely used or known but based on accepted or standard notation

   - *Very Low*: no commonly accepted notation or rules for interpretation; or research work not widely used or known

LAYER	COMPONENT TYPE	ELEMENTS
Function	Functional Description	Operations Data
	Non-functional Req'ts	"-ilities" Timing Security
Design	Domain	
	Architectural Design	Modules Coupling Input Output Processing
	Dependancies	Hardware Software Timing Data Validity
	Alternatives	(same as Arch. Design)
	Rationale	Advantages Disadvantages Decisions
	Process	Methodology Constraints Verification Quality Assurance

Figure 14. Types and Elements of Reusable Software Components

5. Guidelines for evaluating representation *resolution*

   - *Very High*: almost all differences could be represented
   - *High*: all major and most minor differences could be represented
   - *Nominal*: all major and some minor differences could be represented
   - *Low*: most major and some minor differences could be represented
   - *Very Low*: only a few major differences could be represented

Some examples might illustrate the resolution guidelines. For the functionality layer, major differences imply different functions. An example would be the difference between a swap routine that goes through a set of elements swapping their positions, and a sort routine that goes through a set of elements and may swap their positions depending on some ordering scheme. Minor differences in the functionality layer include differences in properties of a function. An example is the difference between a stable sort and an unstable sort. The stability property of a sort function concerns whether the ordering of elements with the same key is maintained (4:464). For the design layer, major differences suggest different approaches or algorithms. An example is the difference between a bubble sort, which uses an element exchange approach, and a heap sort, which uses an element selection approach (4:462). Minor differences in the design layer suggest differences in how an algorithm or approach is implemented. An example is the difference between a bubble sort and a quick sort, both of which are exchange sorts, but one exchanges single elements, and the other exchanges subsets of elements (4:462).

3.2 Representation Evaluation

Given the evaluation criteria and guidelines in Section 3.1, the next step in the process of answering the research questions was to use the criteria to evaluate current

functional and design representations. The representations to be evaluated were those discussed in Chapter II. The evaluation consisted of reviewing documentation describing the representations and evaluating them against the specified criteria, once for the functionality layer and once for the design layer. Once the evaluation was complete, a numeric score was calculated for each technique to arrive at an overall score for each layer. The score was calculated by assigning a numeric value to each possible rating and then averaging the ratings for the criteria for each technique. For this work, the ratings and their corresponding numeric values are shown in Figure 15. The results of the evaluation are discussed in Section 4.1.

Rating	Very Low	Low	Nominal	High	Very High
Value	1	2	3	4	5

Figure 15. Numeric Values of Evaluation Ratings

3.3 Develop Prototype

Once the representations were evaluated, the highest rated representation(s) were incorporated into a prototype reusable software library interface. This involved the substeps described next.

3.3.1 Select Components for Prototype Library One of the main concerns that any library must address is what components should be included in the library. As pointed out earlier by Frakes, the issue is one of costs, including the cost to create and maintain the library (the administrative issue) and the cost in computational resources to use the library (the implementation issue). Clearly the components that are developed and put in the library must be valuable enough that they are worth the cost to include in the library. The prototype developed for this effort was no

exception to this concern. In this case, though, the value of the components was not going to be derived from their subsequent reuse since a prototype is designed to demonstrate a concept, not to become the actual implementation. In this case the value of the components would be derived from their ability to show how effective the layered approach to presenting reusable software components was and how well the evaluation criteria developed earlier identified the most efficient representation. With this in mind, an example set of components was selected to be incorporated into the prototype interface. To simplify the task of developing the prototype while trying to cover most aspects of the representations, the components had to meet the following criteria:

1. Existence: the components were already developed.
2. Familiarity: the component functions were known or recognizable to the group who evaluated the prototype since users would know what functions they were looking for.
3. Similarity: some similar components were included to allow evaluation of the representation resolution.

The following component sets were evaluated for potential use in the prototype: Booch components, Common Ada Missile Packages (CAMP), Ada Software Repository (ASR) components, and graphics library routines. The results of the evaluation are shown in Figure 16. A combination of the Booch components and ASR components were chosen since they were already developed, included basic functions that were familiar, and had several similar components. A combination of the two sets was used since both met the criteria and both had some good example parts. The CAMP parts exist and have similar components, but unless the evaluators had previous experience with missile software, they would be unfamiliar. The graphics library routines already exist and have similar components, but would only be familiar to evaluators with a graphics background.

	Software Components			
Criteria	Booch Components	CAMP Parts	ASR Components	graphics routines
Existence	X	X	X	X
Familiarity	X		X	
Similarity	X	X	X	X

Figure 16. Results of Component Set Evaluation

3.3.2 Prepare Functional and Design Representations of Components Once the representation and the component set were selected, representations of a minimum set of components were developed. For this work, seventeen components were implemented with three sets of similar components. This was to provide a reasonable set of components to view the representations, especially similar components, as a basis for evaluating the strengths and weaknesses of the representations.

3.3.3 Select Interface Technique An interface technique was selected next. The technique was selected based on the following characteristics to simplify the prototype development and use:

1. Simplicity: the interface was easy to use
2. Convenience: the interface had tools or basic constructs to simplify development
3. Graphical Capabilities: the interface was capable of presenting graphical representations
4. Availability: the interface was available easily and at low cost

5. Portability: the interface was able to be demonstrated on as many platforms as possible

Several interface development toolkits were examined, including the Motif X-Windows toolkit, Microsoft Windows Software Development Toolkit and the Simple User Interface Toolkit (SUIT). The results of the evaluation are shown in Figure 17. All three development toolkits provided a simple to use mouse-based graphical user interface. SUIT was rated as convenient since it provided an interface editor that allowed adding or modifying interface elements immediately to the interface. Motif and Windows required explicit and extensive coding to create the interface. All three toolkits allowed for graphical capabilities. Motif and SUIT were available at no charge to educational institutions and were available via anonymous FTP from their respective developers. The Windows Software Development Toolkit cost approximately $400. The SUIT toolkit was available and portable to several different hardware configurations, including Sun Workstations, Sparcstations, Silicon Graphics Iris Workstations, IBM PCs, and R6000 based computers. Motif was only available for X-Windows based computers, and Windows was only available for IBM PC and compatibles. Since SUIT satisfied all five selection criteria, it was selected.

3.3.4 Design and Implement Interface The SUIT interface toolkit was used to implement the prototype interface. The main concern at this point was to create an interface that presented the reusable software components using the layered approach discussed in Section 1.3 and using the highest rated representation technique as described in Section 3.2. The first layer of reuse information was the functionality and the second layer was design information. The last two layers, quality attributes and source code, were not specifically addressed because the emphasis was on the representation of functionality and design, but they were included since they were not difficult to implement. This work was not concentrating on database issues such as

	Toolkits		
Criteria	Motif	MS Windows	SUIT
Simplicity	X	X	X
Convenience			X
Graphical Cabability	X	X	X
Availability	X		X
Portability			X

Figure 17. Results of Interface Development Toolkit Evaluation

efficient storage or retrieval so the components were not put into a formal database. The representations were implemented simply, using the graphical capabilities of the SUIT toolkit and accessed using a faceted keyword retrieval approach similar to the one presented in (31). The reason for this approach was to stay within the time constraints of this project.

3.4 Develop Questionnaire

A questionnaire was developed to get informal feedback on the strengths and weaknesses of the component representations and layered approach used in the prototype. The questions were developed to be specific about the user's experience since these types of questions provided more useful results (24:896). The questions were composed to cover each criterion used in selecting the representation to see how well the representation met the criteria as well as how important each criterion was perceived by the people using the prototype. Finally, general questions were included to get an idea of the specific background of the people who did the evaluation.

3.5 Evaluate Prototype

A group of software engineering professionals was selected to evaluate the prototype. The evaluators came from a wide background within the Air Force. The evaluators were given a short training lesson on the prototype interface and then were given time to explore the prototype and the representations. The questionnaire was given to the evaluators as a guide to direct their review.

3.6 Analyze Results

After the group of evaluators finished their critique of the prototype, the questionnaires were reviewed to see if there were any areas identified as weaknesses or areas suggested for improvements. These suggestions and problem areas were summarized as recommendations for improvement and are presented in Section 5.7.

IV. Results

4.1 Representation Evaluation Results

The representation techniques discussed in Chapter II were evaluated using the evaluation criteria presented in Section 3.1.1. The guidelines in Section 3.1.2 were used to assign ratings to each representation technique. Each technique was evaluated for both the functionality and design layers. A summary of the evaluation for the functional layer is shown in Figure 18. A summary of the evaluation for the design layer is shown in Figure 19. The main differences in the ratings for the different layers could be seen in the expressiveness and resolution ratings. Some techniques express component functionality very well and were rated highly for expressiveness in the functional layer, while they do not express design issues very well and so were rated much lower for expressiveness in the design layer. A few techniques express both layers fairly well. As for resolution, some techniques could show differences between similar components at the functional level very well, while others did not. Most techniques could show differences better at the functional level than at the design level because there were fewer component types to try to represent.

Hypertext and a metamodel were evaluated separately using the same criteria. The evaluation criteria, however, were redefined slightly to take into consideration the broader perspective of these techniques and their ability to show multiple views. The definitions used for the criteria in these evaluations were:

1. *Generality:* this criterion examined what limitations existed on the type of views that could be presented.
2. *Expressiveness:* this criterion examined how complete was the picture of the component that was provided by the multiple views.

	Criteria				
Representation	Gen	Exp	Und	Con	Res
Text Description	VH	VH	VH	N	VH
Keywords/Facets	VH	H	VH	N	N
Frames	N	H	L	N	H
Forms	L	VH	H	N	H
Formal/Logical	VH	H	L	H	N
DFD	VH	N	H	H	L
Semantic Nets/E-R	N	N	N	L	L
PDL	H	VH	VH	N	H
Structure Charts	N	L	N	N	L
Plan Calculus	VH	VH	L	L	H
Schemas	VH	N	N	N	L

KEY:

Gen - generality	VH - very high
Exp - expressiveness	H - high
Und - understandability	N - nominal
Con - consistency	L - low
Res - resolution	VL - very low

Figure 18. Functional Layer Representation Evaluation Results

	Criteria				
Representation	Gen	Exp	Und	Con	Res
Text Description	VH	L	VH	N	N
Keywords/Facets	VH	VL	VH	N	VL
Frames	N	N	L	N	L
Forms	L	H	H	N	H
Formal/Logical	VH	L	L	H	L
DFD	VH	VL	H	H	N
Semantic Nets/E-R	N	L	N	L	VL
PDL	H	H	VH	N	H
Structure Charts	N	L	N	N	N
Plan Calculus	VH	VH	L	L	H
Schemas	VH	H	N	N	L

KEY:

Gen - generality	VH - very high
Exp - expressiveness	H - high
Und - understandability	N - nominal
Con - consistency	L - low
Res - resolution	VL - very low

Figure 19. Design Layer Representation Evaluation Results

3. *Understandability:* this criterion examined whether the relationships between the views could be easily comprehended by people.

4. *Consistency:* this criterion examined how well the views were integrated together.

5. *Resolution:* this criterion examined how much detail could be captured by the multiple views.

The results of their evaluation are shown in Figure 20. The rationale for all the evaluations is included in Appendix A.

	Criteria				
Representation	Gen	Exp	Und	Con	Res
Hypertext	VH	H	H	N	VH
Metamodel	H	H	N	H	N

KEY:

Gen - generality	VH - very high
Exp - expressiveness	H - high
Und - understandability	N - nominal
Con - consistency	L - low
Res - resolution	VL - very low

Figure 20. Hypertext and Metamodel Evaluation Results

Once the ratings were assigned, a numeric score was calculated for each technique by averaging the numeric values of the ratings, shown previously in Figure 15, for each technique. The scores for each technique are shown in Figure 21. As shown in the figure, the simple text description obtained the highest overall score for the functionality layer. At the design layer, PDL received the highest rating, followed closely by text descriptions and plan calculus, and then forms and schemas.

Representation	Score	
	Functional Layer	Design Layer
Text Description	4.6	3.6
Keywords/Facets	4.0	3.0
Frames	3.2	2.6
Forms	3.6	3.4
Formal/Logical	3.6	3.0
DFD	3.6	3.4
Semantic Net/E-R	2.6	2.2
PDL	4.2	4.0
Structure Charts	2.6	2.8
Plan Calculus	3.6	3.6
Schemas	3.2	3.4

Figure 21. Numeric Scores of Representation Technique Evaluation

4.2 Research Question 1 Results

The first research question presented in Section 1.4 was "how is software functionality currently represented?" Several representations were presented in Section 2.1 to answer this question. These representations included both textual and graphical methods. The representations had their origins in the areas of language (text descriptions and forms), information classification and retrieval (keywords and facets), database technologies (entity-relationship diagrams), artificial intelligence (semantic nets and frames), mathematics (formal and logical approaches), and software development (data flow diagrams). These areas are general groupings that cover most of

the types of software representations (35:8-9). The techniques that were discussed were representative of how software functionality is currently represented.

4.3 Research Question 2 Results

The second research question presented in Section 1.4 was "how can software functionality be presented to a user?" This question was answered in part by the evaluation of the representations presented in Section 2.1. The results of the evaluation were shown in Figure 18. The results were that the highest rated representation for software functionality was simple text descriptions. To complete the answer to the question, a prototype reusable software library was developed to present the software component information using the layered approach discussed in Section 1.3. The functionality layer of the prototype was implemented using text descriptions and included some information suggested by Frakes and Nejmeh appropriate to describing functionality (19:147). The actual description included the name of the component, author, creation date, a short abstract, keywords, properties, inputs, and outputs. The prototype was evaluated to determine how well the software functionality was presented to the user using this approach. The results of the evaluation are discussed in Section 4.6.2.

4.4 Research Question 3 Results

The third research question presented in Section 1.4 was "what methods are effective in representing reusable software component design and related design information?" Several representations were presented in Section 2.2 to answer this question, including textual methods and graphical methods. The representations were evaluated using the criteria outlined in Section 3.1.1 to determine which ones were effective. The results of the evaluation were shown in Figure 19, with the averaged numeric scores shown in Figure 21. The highest rated representation was

Program Design Language (PDL), closely followed by text descriptions and Plan Calculus. The score for Plan Calculus was the same for the design layer as the functionality layer, but it was one of the highest scoring representations for the design layer because most other representations had lower scores for the design layer than the functionality layer. The reasons for these results are discussed in Section 5.4. The close scores for several representations showed that there were several techniques that were about equally effective in representing reusable software component design.

4.5 Research Question 4 Results

The fourth research question presented in Section 1.4 was "how can software designs and design information be presented to a user?" A prototype reusable software library was developed to present the software component information using the layered approach discussed in Section 1.3. The design layer of the prototype was implemented using examples of the three techniques that had the highest average numeric scores from Figure 21. These were PDL, Plan Calculus and text descriptions. Most of the component representations used PDL as specified in (8) to show the architectural design. Then some components were also implemented using the Plan Calculus as described in (33). The components chosen to be implemented in the Plan Calculus were similar to each other so that they could be compared to each other in both the Plan Calculus and the PDL format. Finally all the components also had an associated text description for the design information. The prototype was evaluated to determine how well the software design information was presented to the user using these various approaches. The results of the evaluation are discussed in Section 4.6.2.

4.6 Prototype Reusable Software Library

4.6.1 Prototype Development The prototype was developed using the SUIT toolkit and layered approach as suggested in Section 3.3.4. An interactive mouse and menu driven interface was developed using the capabilities of the SUIT toolkit. The interface was comprised of menus to save and retrieve information stored in files, to specify and revise component search criteria, to view the various layers of component information, and to present miscellaneous utilities and help information. The purpose of the interface was to simplify the user's task of searching for and viewing a set of software components at various levels of detail.

The interface was developed to allow the addition of components to the database with minimal impact to the interface code. This required splitting the interface into two programs: a program to present the interface menus, request the search criteria, and allow selection of matched components; and a program to draw the selected component information. The interface program simply reads and searches a component information file that is a separate text file which could be edited without affecting the interface program. The drawing procedures for the components, however, currently have to be coded as part of the drawing program that must be recompiled each time a component is added. In this way the main interface was kept separate from the details of the component representations and could be used with any component representation.

A demonstration script, or tutorial, was written to familiarize the prototype evaluation participants with how to use the prototype. It included a description of the concept behind the layered approach to presenting the reusable software component information as well as a description of how the interface worked. The description of the interface described each menu that was available and their associated commands. It then presented several step-by-step procedures to perform the common

actions associated with finding a reusable component from the library. Each procedure also included a running example to describe what the expected results were for each action. The tutorial that was provided to the evaluators is included as Appendix B. Some figures showing examples of how the interface appeared are included in Appendix C.

4.6.2 Prototype Evaluation A small set of evaluators was selected to complete the prototype evaluation. A copy of the questionnaire that was provided to the evaluators is included in Appendix D. The purpose of the evaluation was to provide a "dry run" of the prototype and questionnaire, to see where any difficulties might arise or where some improvements and clarifications could be made. Seven software engineers with varied software development experience completed the evaluation. A summary of their responses is included in Appendix E. The scale used for the evaluation ranged from 1 to 5, which corresponded to the responses: strongly disagree, somewhat disagree, borderline, somewhat agree, and strongly agree, respectively.

Most of the evaluators agreed with the statements about the functionality layer representation, with responses between 4 (somewhat agree) and 5 (strongly agree). The statements with the highest agreement included that the functionality representation was independent of specific programming languages, development methodologies, and applications; the representation specified what the component did; and the representation was understandable and unambiguous (average responses from 4.3 to 4.9). The two statements with the lowest averages and most varied responses were how well non-functional requirements were represented (average response 3.8, standard deviation 1.2) and how well the representation distinguished between similar items (average 3.4, standard deviation 1.4). Some of the comments provided for the functionality layer asked for the definition of non-functional requirements (as

used in statement 5), asked for the definition of properties (a keyword used in the representation,) and asked how exceptions and errors were represented.

The responses for the design layer representations (without text supplement) were much more varied. The PDL representations had higher agreement on the average than the Plan Calculus representations. The statements that had the lowest responses for both representations were how well the representations specifed system dependancies, design alternatives, design rationale, and design methodology. These were statements 18 through 21 for PDL (with average responses from 2.4 to 2.9) and the corresponding statements 38 through 41 for Plan Calculus (averages from 2.1 to 2.6). These statements also had the most variations in responses, with standard deviations for the PDL representation ranging from 1.2 to 1.5 and for the Plan Calculus representation ranging from 1.4 to 1.5. The Plan Calculus representation also had consistently lower responses for statements addressing whether the representation was understandable, unambiguous, distinguished between similar items, and specified architectural design, with average responses from 2.3 to 3.6. Some of the comments, however, indicated that the evaluators were unfamiliar with the representation and that it was very difficult to comprehend. One comment suggested an improvement by having a help screen that included definitions of the Plan Calculus symbology to explain what they meant.

The responses for the text supplement for the design layer were similar between the PDL and Plan Calculus representations, but varied in degree. The statement that the text was necessary to understand the component representation (statement 23 and 43) was highly agreed with for Plan Calculus (average 4.6, std dev .73), but only moderately agreed with for PDL (average 3.6, std dev .49). The statement that text was necessary for distinguishing between similar components (statement 24 and 44) was only moderately agreed with for both PDL and Plan Calculus (average

3.6 and 3.7, resp.) Text was not viewed as necessary for representing architectural design (statements 25 and 45) for either PDL or Plan Calculus (average 2.0 and 3.0 resp.) The statements that text was necessary to represent system dependancies, design alternatives, design rationale, and design methodology (statements 26 - 29 and 46 - 49) were agreed with to varying degrees for the two representations (PDL average responses from 3.7 to 4.4 and Plan Calculus average responses from 3.6 to 4.6). One of the evaluators made that comment for the PDL representation that "it is important to use both the PDL and text in making decisions."

Finally, the statements concerning the prototype interface were consistently agreed with. The three statements that the layered approach was helpful to understanding components, the interface was easy to use, and the interface was logically arranged (statements 53, 56, and 57), were all strongly agreed with by all the evaluators (response 5). The statements that the layered approach was helpful for comparing components and the layered approach covered component information well were also agreed with (average responses 4.9 and 4.6, respectively.) The comments provided about the interface indicated that the interface was liked, that it was "simple, yet effective."

V. Conclusions and Recommendations

5.1 Representation Evaluation Conclusions

The representation evaluation revealed that one of the evaluation criteria may not be appropriate. The evaluation criterion that provided questionable value was *expressiveness*, which was defined as how well the representation technique could represent different types of components for the various layers of information. Most representations examined were not designed to represent several types of information, so most of them had low expressiveness ratings. Since this was a result of the definition and it affected most of the representations the same way, it did not add much to the evaluation. One conclusion may be that the definition was inappropriate and one representation should not be expected to represent everything. In this case a reusable software library may require a couple of different types of representations for different types of information, such as the combination of graphical and textual presentations used in the prototype. The other conclusion may be that a new representation still needs to be developed that can represent several types of information. Arguments could be made for both cases.

A second evaluation criteria that had consistently low results was *resolution*. Resolution was defined as indicating how well the representation could represent small differences between similar components. This characteristic, however, is important for a reusable library. The implications of the low resolution ratings are further discussed in Sections 5.2 and 5.4.

5.2 Research Question 1 Conclusions

Several representations were presented in Section 4.2 to answer the question "how is software functionality currently represented?" Each of the representations

was evaluated as described in Section 4.1. The areas where most of the representations were weak were consistency (average rating of "nominal") and resolution (average rating of "nominal"). The weak consistency rating showed that most of the current representations for software functionality were not standardized or widely used yet. No one has developed a representation that is sufficient to meet the needs and expectations for representing functionality. This is also reflected by the fact that most representations did not have widely available automated tool support. They lacked the characteristic of machine processability.

The weak resolution rating indicated that most current representations were not expressive enough to allow users to easily distinguish between similar components for a software library environment. As a result, most current libraries either offer no help in choosing a component and require the user to examine source code to make a decision, or provide artificial help in selecting components using arbitrary evaluation schemes. One such approach was to provide an abstract ranking of components based on assumed relationships between descriptive keywords (31:11-12). So while several representations for software functionality exist, none is currently the best answer to representing software components.

5.3 Research Question 2 Conclusions

The representation evaluation and the prototype were used to try to answer the question "how can software functionality be presented to a user?" As presented in Section 4.1, text description was the highest rated functionality representation in the evaluation. This was due in large part because non-functional component requirements are currently difficult to represent without text. These requirements include reliability, maintainability, safety, timing, security, concurrency and similar requirements. Most representations reviewed did not include any method of

representing the non-functional aspects of component functionality. Some of these characteristics, however, would be found in the third representation layer, which is quality attributes. But this would make it harder for a user trying to find a component with a specific function and timing constraint, for example. They would have to view two different layers rather than just one. Plan Calculus did recognize the need for additional types of information and allowed constraints to be included in the representation as predicate calculus assertions. The main shortcomings of Plan Calculus, though, were that it required specific training to understand the notation (low understandability) and it was not a standardized or widely used notation (low consistency).

The prototype evaluation provided some additional perspectives about representing software functionality to a user. The questionnaire presented several statements about the functionality representation, which was implemented using text descriptions based on the results of the representation evaluation mentioned above. The two statements from the questionnaire that had the lowest average responses with the most variations were how well non-functional requirements were represented and how well the representation distinguished between similar items. These results may indicate areas that require improvement in the representation. They may also indicate areas where clarifications in the prototype are required, since some of the comments from the questionnaire asked for the definition of non-functional requirements (from statement 5) and the definition of properties (a keyword used in the prototype functionality representation.) A re-evaluation would need to be performed after the clarifications were added to the prototype to determine whether the representation needed further improvement.

Based on the observations mentioned, a new representation apparently is still needed or an existing representation should be modified to address the issues of

representing non-functional requirements and distinguishing between similar components. The representation then needs to become widely accepted or standardized. Until a representation is developed that can adequately address these issues, text will remain the most common representation for functionality, or at least will be a necessary supplement to any representation.

5.4 Research Question 3 Conclusions

The design representation evaluation was used to answer the question "what methods are effective in representing reusable software component design and related design information?" The text based techniques were again among the highest rated design representations, with PDL, text descriptions, and forms having three of the highest average numeric scores. Again the ratings were influenced by some design components that are more easily expressed in text, such as design dependencies, rationale, and design process elements such as methodology, verification, and quality assurance.

One unexpected result, however, was that most of the representations had lower scores for the design layer than the functionality layer. This was because even though the ratings for the criteria *generality, understandability,* and *consistency* were largely independent of the representation layer and did not change between the two layers, the ratings for *expressiveness* and *resolution* were mostly lower for the design layer than the functionality layer. The average expressiveness and resolution ratings were both between low and nominal. As mentioned above, the ratings were influenced partly by the fact that the design layer was defined to include several types of design information in addition to the commonly thought of architectural design. Most of the representations reviewed did not include any method of representing these additional aspects of component design. As a result, most of the representations

were rated low for expressiveness. The weak resolution ratings again indicated that most current representations were not expressive enough to allow users to easily distinguish between similar design components for a software library environment. The generally lower scores for the design representations and closeness of several of the scores indicated that no representation was distinctly more effective than the others for representing design information.

5.5 Research Question 4 Conclusions

The prototype was used to answer the question "how can software designs and design information be presented to a user?" The prototype design information was implemented using a layered approach with PDL, Plan Calculus, and text descriptions as described in Section 4.5. The text descriptions were added to the design representations to provide clarifications of design information that could not be easily captured by the PDL and Plan Calculus.

The prototype evaluation revealed several areas that could be improved for the design representation, as discussed in Section 4.6.2. For the Plan Calculus representations, the lower responses for the statements on representation understandability, ambiguity, distinguishability, and descriptiveness indicate that Plan Calculus has several hurdles to overcome before it could be an acceptable design representation for a reusable component library. It would either have to become standardized so that it is commonly taught as a representation so that many people would understand it, or the library itself would have to provide a very detailed tutorial on the Plan Calculus symbology so users could determine what a particular component representation meant. The first approach would be better since a standard would help insure that the representations were interpreted consistently by different people. The second approach of having a tutorial available would also be beneficial even

if the representation was standardized to allow users to have on-line help available while viewing component representations. A tutorial without having a standardized representation would only be of limited value.

The statements that had the lowest responses for both PDL and Plan Calculus representations addressed how well the representations specifed system dependancies, design alternatives, design rationale, and design methodology. This supports the results of the representation evaluation discussed in Section 5.4 that the representations were not designed to represent many different types of design information. The implications of these results were discussed in Section 5.1. The responses to the statements about the text supplement to the design information indicated that the text was not required to be present, but that it was very helpful, especially for the Plan Calculus representations that may not have been familiar to the user.

The results from the prototype evaluation indicated that no representation was definitely more effective than the others for representing design information. For the current representations, text description was a useful supplement to the representation. The biggest issue seemed to be trying to represent several types of design information. All these items imply that a representation needs to be developed or an existing one modified to allow representing many types of design information while being easy to understand and widely used.

5.6 Prototype Conclusions

One area of the prototype that was weak was the drawing of the representations. The representations were drawn by a program that was developed using the SUIT drawing capabilities, which were somewhat limited. The SUIT graphics functions were limited to lines, arcs, rectangles, and text. A function had to be coded for each higher level object used in the various representations, such as the hatched

lines and lines with arrows in the Plan Calculus representations. The representations were also constrained to being developed as part of the drawing program. Any changes to the representations had to be recoded and the entire drawing program recompiled. This was acceptable for the prototype, but for an actual reusable library system would be unacceptable. This would be improved if the representations had automated tools that could draw the representation independently from the drawing program. The library system would then be much more portable and would require much less maintenance as the component collection grew. The representations used in the prototype, however, were not standardized or the most widely used, with the resulting lack of tool support and reflected low to nominal consistency ratings.

The prototype evaluation provided additional insights about the prototype, through explicit comments as well as observations from the results. One comment was made that it might be helpful if the user had the choice of representations so they could pick the one they understood the best. Another evaluator stated that it might be more helpful to have available the text information, PDL representation *and* the graphical representation, such as the Plan Calculus, available to present the component information. The responses from the questionnaire also indicated that the representations used were not the best representations, as discussed in the previous sections. Most of the comments about the interface itself, including menus, arrangement, and so on, indicated that the interface worked well, and the layered approach to presenting component information was successful.

5.7 Recommendations for Further Work

One area that could be followed up is to incorporate the clarifications and improvements for the prototype suggested in the questionnaire and mentioned above. Part of the purpose of the prototype evaluation "dry run" was to find the areas that

needed more work in the prototype so they could be eliminated. Once these areas were identified, the next step would be to include them in the prototype. Then a more thorough investigation and evaluation could be performed. The results of a second evaluation would more accurately reflect the effectiveness of the representations evaluated and not be influenced as much by shortcomings in the prototype.

A second area that could be pursued farther is in evaluating additional functional and design representations. The purpose would be to find some that better meet the evaluation criteria, to provide a stabler basis for evaluating the effectiveness of the layered approach to presenting reusable component information. The representations included in this effort were only representative of the many techniques available. New ones are continually being researched and developed. Some new ones have begun to address the problems pointed out in the evaluation, such as the work done with Plan Calculus and formal representations. A new technique may be available that addresses the issues raised here. Only after a more thorough investigation has been performed and a more detailed analysis undertaken could a definitive answer be made to the question of what is the best representation method for a reusable software library. This work was mainly to develop the presentation concepts and evaluation criteria that lead to the next step.

Another area of work that could be pursued is adding to the component library. The components included in the prototype were chosen to simplify the development of the prototype as well as to help in the evaluation of the representations. A larger component collection with more realistic application components may give more useful evaluation results. One approach to expanding the component collection would be to find an appropriate domain, perform a domain analysis, and develop reusable components that implement the commonalities found between systems in that application domain (12:224). This approach would allow the systematic development

of the information in the functionality layer, design layer, and quality layer as well as the final source code. A second approach would be to reverse engineer existing components that are widely usable. This approach would have the advantage of having source code that is immediately available for inclusion in the library. The higher levels of information could be added as time and resources permitted. Either approach would work in expanding the component library.

Appendix A. *Rationale for Representation Evaluations*

A.1 Rationale for Functional Layer Evaluations

The following sections provide the rationale for why the various representations were rated as they were for the component functionality layer.

A.1.1 Evaluation of Textual Descriptions The following evaluations are for textual descriptions as a representation of software components. The descriptions are assumed to be in simple English prose and do not include any programming language constructs or actual source code. Those constructs would be included under Program Design Language (PDL).

1. *Generality* (rated VH): since ordinary text is used, it is not limited to using any common or specific constructs or terms.
2. *Expressiveness* (rated VH): the functionality elements (objects and operations) are directly expressed by the representational elements nouns and verbs, and the non-functional requirements could be expressed by adjectives and simple combinations of nouns and verbs.
3. *Understandability* (rated VH): the only knowledge required is common reading skills and general knowledge of software.
4. *Consistency* (rated N): English has rules for interpretation, but since words have several shades of meanings, the representation can be ambiguous.
5. *Resolution* (rated VH): text could be used to describe functional and non-functional elements very well.

A.1.2 Evaluation of Keywords/Facets The following evaluations are for keywords, or facets, as a representation of software components. It is assumed that the keywords are not constrained to any particular language, methodology or applica-

tion. While the number of keywords was not specifically assumed, it was assumed the number was limited.

1. *Generality* (rated VH): since keywords and facets are chosen from ordinary terms, they are not limited to common or specific language, methodology or application constructs or terms.

2. *Expressiveness* (rated N): the non-functionality elements would require complex combinations of keywords to represent and would be limited by number of keywords.

3. *Understandability* (rated VH): they are not constrained to common or specific language, methodology or application constructs or terms.

4. *Consistency* (rated N): keywords could have various shades of meanings, which could allow ambiguity.

5. *Resolution* (rated N): only major and a couple minor differences in functional and non-functional elements could be represented since only a few key words are used.

A.1.3 Evaluation of Frames The following evaluations are for frames, as described in (15), as a representation of software components.

1. *Generality* (rated N): it used a few methodology specific terms such as object, attribute and action (operation) (15:41).

2. *Expressiveness* (rated H): it had elements (slots and fillers) to represent functional and non-functional aspects with minor inconsistencies.

3. *Understandability* (rated L): the frames use specific constructs which would require training (e.g. has-actor, has-agent, has-environment (15:44)).

4. *Consistency* (rated N): frame-based approaches have some variations

5. *Resolution* (rated H): the frames could represent major differences and some minor differences with a little work.

A.1.4 Evaluation of Forms The following evaluations are for forms, as described in (27), as a representation of software components.

1. *Generality* (rated L): it used Ada language specific constructs such as task, private and exception (27:173-75).

2. *Expressiveness* (rated VH): they have elements (Ada elements) to represent functional elements, and can use text to represent non-functional elements

3. *Understandability* (rated H): since the forms use Ada as well as text, they would require some general Ada knowledge to understand.

4. *Consistency* (rated N): forms are essentially a known notation (Ada) with some extensions.

5. *Resolution* (rated H): forms could represent most major differences and some minor differences with some work.

A.1.5 Evaluation of Formal/Logical Approaches The following evaluations are for formal or logical representations for software components.

1. *Generality* (rated VH): it used mathematical or logical notation which was not constrained to any language, methodology or specific application.

2. *Expressiveness* (rated H): it can represent functional elements, but needs combinations to represent non-functional elements.

3. *Understandability* (rated L): formal notations such as Ted (20) and Z (22) require some training or instruction to understand and interpret them.

4. *Consistency* (rated H): they are based on extensions to known mathematical or logical notation, but some variations exist.

5. *Resolution* (rated N): most major differences in functional elements are representable, but minor differences would be difficult.

A.1.6 Evaluation of Data Flow Diagrams The following evaluations are for Data Flow Diagrams (DFDs) as a representation of software components.

1. *Generality* (rated VH): the graphical elements of DFDs are not constrained to any language, methodology or application constructs or terms.

2. *Expressiveness* (rated N): it has elements for the functional elements, but would require complex combinations to represent non-functional elements.

3. *Understandability* (rated H): DFDs have only simple graphical elements that are commonly taught as part of most software development courses.

4. *Consistency* (rated H): DFDs have a commonly accepted notation with some minor variations.

5. *Resolution* (rated L): most major differences in functionality could be represented, but few minor differences in functional or non-functional elements.

A.1.7 Evaluation of Semantic Nets/E-R Diagrams The following evaluations are for semantic nets and entity relationship diagrams as representations of software components.

1. *Generality* (rated N): they use the specific methodology terms and constructs of objects and attributes (16:1432).

2. *Expressiveness* (rated L): it has elements (objects) for the functional element data, but would require complex combinations to try to represent operations and non-functional elements.

3. *Understandability* (rated N): E-R diagrams are a common represenation included in general sofware development education, but semantic nets are not as common.

4. *Consistency* (rated L): E-R diagrams have common notation with a few variations and extensions, while semantic nets have a few variations but little agreed upon semantics (16:1432).

5. *Resolution* (rated L): few non-functional elements could be represented.

A.1.8 Evaluation of PDL The following evaluations are for PDL as a representation of software components. PDL is assumed to not be constrained to any one language, but may include actual source code.

1. *Generality* (rated H): PDL uses only text and common language constructs and terms.

2. *Expressiveness* (rated VH): PDL has the same elements available as text to represent functional and non-functional elements.

3. *Understandability* (rated VH): PDL requires only general knowledge of common programming language constructs and terms.

4. *Consistency* (rated N): PDL has commonly accepted notation using common language constructs, but has several variations.

5. *Resolution* (rated VH): same effectiveness as normal text description.

A.1.9 Evaluation of Structure Charts The following evaluations are for Structure charts as a representation of software components.

1. *Generality* (rated N): constrained to use common language constructs such as function call and iteration.

2. *Expressiveness* (rated L): it has elements for functional elements, but can't represent non-functional elements.

3. *Understandability* (rated N): some training in software development and representations required.

4. *Consistency* (rated N): has a commonly accepted notation with various extensions.

5. *Resolution* (rated L): functional elements could be represented with simple combinations, but non-functional elements would not be usefully represented.

A.1.10 Evaluation of Plan Calculus The following evaluations are for Plan Calculus as a representation for software components.

1. *Generality* (rated VH): not constrained by common or specific language, methodology or application constructs or terms.

2. *Expressiveness* (rated VH): it has elements for all functional and non-functional elements

3. *Understandability* (rated L): requires some training to understand new notation.

4. *Consistency* (rated L): research work not widely used but built on known concepts of data flow and predicate calculus (34:323-324).

5. *Resolution* (rated H): most elements could be simply represented.

A.1.11 Evaluation of Schemas The following evaluations are for Schemas as a representation for software components.

1. *Generality* (rated VH): not constrained by common or specific language, methodology or application constructs or terms.

2. *Expressiveness* (rated N): it has elements for functional elements, but would require complex combinations to express non-functional elements.

3. *Understandability* (rated N): only general training with software representations required since Schemas use DFDs.

4. *Consistency* (rated N): they have a common notation with some addtional extensions, since schemas are based on DFDs.

5. *Resolution* (rated L): could not easily represent non-functional elements, such as timing or resources required.

A.2 Rationale for Design Layer Evaluations

The following sections present the rationale for why the various representations were rated as they were with respect to representing component design information. Since the evaluation criteria *generality, understandability,* and *consistency* were largely independent of the representation layer and did not change between the two layers, they are not repeated here. Only the criteria that changed are shown below.

A.2.1 Evaluation of Textual Descriptions The following evaluations are for textual descriptions as a representation of software components. The descriptions are assumed to be in simple English prose and do not include any programming language constructs or actual source code. Those constructs would be included under Program Design Language (PDL).

1. *Expressiveness* (rated L): it requires extensive (lengthy) text description to represent design elements, compared to a more succinct graphical representation, for example.

2. *Resolution* (rated N): text could be used to describe functions and algorithms well enough to show differences between components, but few low level implementation differences could be described.

A.2.2 Evaluation of Keywords/Facets The following evaluations are for keywords, or facets, as a representation of software components. It is assumed that the keywords are not constrained to any particular language, methodology or application. While the number of keywords was not specifically assumed, it was assumed the number was limited.

1. *Expressiveness* (rated VL): several elements would not be representable using just a few keywords, such as design rationale, test plans and procedures; and others, such as design and algorithms, would be difficult to represent.

2. *Resolution* (rated VL): only a few major differences in function and algorithms could be represented, since only a few key words are used.

A.2.3 Evaluation of Frames The following evaluations are for frames, as described in (15), as a representation of software components.

1. *Expressiveness* (rated N): frames would require some simple and some complex combinations of slots and filler to represent the design elements.

2. *Resolution* (rated L): the frames could represent most major differences, but would be difficult to represent low level differences.

A.2.4 Evaluation of Forms The following evaluations are for forms, as described in (27), as a representation of software components.

1. *Expressiveness* (rated H): forms could represent most design elements but would need some longer descriptions to represent validation information.

2. *Resolution* (rated H): The forms could represent most differences for Ada components, but may have difficulty for components in other languages.

A.2.5 Evaluation of Formal/Logical Approaches The following evaluations are for formal or logical representations for software components.

1. *Expressiveness* (rated L): it would be difficult to represent low level elements such as design alternatives or rationale and would require complex combinations of elements to represent validation information.

2. *Resolution* (rated L): differences in algorithms are representable, but low level implementation details are not.

A.2.6 Evaluation of Data Flow Diagrams The following evaluations are for Data Flow Diagrams (DFDs) as a representation of software components.

1. *Expressiveness* (rated VL): it could not usefully represent design elements such as architectural design, rationale, or validation information.

2. *Resolution* (rated N): different design approaches could be represented, but few low level implementation details.

A.2.7 Evaluation of Semantic Nets/E-R Diagrams The following evaluations are for semantic nets and entity relationship diagrams as representations of software components.

1. *Expressiveness* (rated L): it is difficult to represent architectural design elements and validation information.

2. *Resolution* (rated VL): only a few major differences could be represented, almost no low level implementation details.

A.2.8 Evaluation of PDL The following evaluations are for PDL as a representation of software components. PDL is assumed to not be constrained to any one language, but may include actual source code.

1. *Expressiveness* (rated H): PDL has elements for architectural design, and with some work expresses design rationale and validation information.

2. *Resolution* (rated H): almost all differences could be represented, but minor differences would require some work.

A.2.9 Evaluation of Structure Charts The following evaluations are for Structure charts as a representation of software components.

1. *Expressiveness* (rated L): it has elements for architectural design, but couldn't usefully represent rationale or validation information.

2. *Resolution* (rated N): low level differences are difficult to represent.

A.2.10 Evaluation of Plan Calculus The following evaluations are for Plan Calculus as a representation for software components.

1. *Expressiveness* (rated VH): has graphical elements for design aspects, with some work to represent validation information and architectural design.

2. *Resolution* (rated H): most major differences could be represented, with some work required to represent minor differences.

A.2.11 Evaluation of Schemas The following evaluations are for Schemas as a representation for software components.

1. *Expressiveness* (rated L): would require complex combinations of data flows and constraints to represent elements such as rationale and validation information.

2. *Resolution* (rated L): could not represent some low level differences, such as timing or resources required.

A.3 Rationale for Hypertext and Metamodel Evaluations

A.3.1 Evaluation of Hypertext The following evaluations are for hypertext as part of the representation of software components. The nodes of the hypertext are assumed to be able to include any text or graphics, including other representation techniques.

1. *Generality* (rated VH): hypertext is not constrained on what type of views it can include as nodes, assuming it is capable of graphical links.

2. *Expressiveness* (rated H): it can provide many views to create a complete picture, but has no guidance or criteria to ensure it.

3. *Understandability* (rated H): it does not require any special knowledge to follow the links which simply "point" to more information.

4. *Consistency* (rated N): No enforced consistency between views, simply jumps — the user can get lost.

5. *Resolution* (rated VH): hypertext can capture as much detail as needed.

A.3.2 Evaluation of a Meta-model The following evaluations are for a Meta-model as described in (23) as a representation of software components.

1. *Generality* (rated H): there are no explicit constraints on what views could be included, but currently only three are defined.

2. *Expressiveness* (rated H): it provides the main views of objects and relationships, data and operations, and states and transistions.

3. *Understandability* (rated N): it requires some specific training and experience to understand how all three methods are tied together.

4. *Consistency* (rated H): the three views have been related and described how they tie together.

5. *Resolution* (rated N): limited to the resolution of the three views.

Appendix B. *Tutorial for the Prototype Reusable Software Library*

The demonstration must be run from Open Windows, version 3. To start the demo, change to the psiebels/suit/thesis directory from the command line or file manager, if not already there. Then enter *proto* at the command line or click on the PROTO executable file icon in the file manager.

B.1 Overview

The main window is divided into two areas: the menu bar and the work area. The menu bar has the following pulldown menu options: File commands, Search commands, View commands, Utilities, System commands and Help. It also has a fast quit button in the upper left corner. The actual component information is presented in another window that will appear on the screen. If you don't know how to interact with Open Windows, ask.

The following are short descriptions of the various menus.

- File Commands: These include commands to save the current search criteria to a file, retrieve previous search criteria from a file, and exit the prototype. The save and retrieve functions are currently not implemented.
- Search Commands: These include commands to specify a new search criteria or revise the previous search criteria. If no previous search criteria was specified, an information message is shown.
- View Command: This function allows you to view a component from a set of components found for a previous search. If no search has been performed yet, an information message will so state.

- Utilities: This menu provides some component utilities to print a selected source file or save it to disk. These utilities have not been implemented in the prototype.
- System Commands: This menu provides functions to add and delete components from the library. These functions are restricted and require a password.
- Help: This function presents a dialog box listing information about the menu choices and some definitions used in the prototype.

B.2 Demonstration

The following sections present the various features that will normally be used with the prototype. Each section lists a step by step procedure on how to normally perform the desired task. The sub-bullets describe what should be displayed as a result of each step using an example.

B.2.1 Helpful Information

B.2.1.1 Prototype Concept The Prototype Reusable Software Library Utility was developed using a layered approach to presenting reusable software component information. The components are presented in four layers: a high level description of functionality, a detailed description of design information, a list of quality attributes, and finally the source code. One of the concepts behind the layered presentation is to show the component information at various levels of detail for many similar components to allow comparison between the components. This is similar to the way common electronic hardware is presented in a Transistor-Transistor Logic (TTL) data book. The prototype is structured to allow the information to be presented in this manner. When a list of components is found, the components can

be viewed and compared at any of the four levels of detail. The rest of this tutorial explains how to accomplish this.

B.2.1.2 Graphical Interface The graphical interface consists of boxes and windows that appear on the screen to provide information or request a response. These boxes are called "dialog boxes". They can present any type of information, but will always have at least two buttons, *OK* and *Cancel*, that are used to indicate when the user is done interacting with the dialog box. *OK* indicates a normal end to the dialog, while *Cancel* indicates that the user does not want anything described in the dialog box to continue.

The mouse is the main interface to the prototype. Most interaction takes place by responding to the dialog boxes using the dialog box buttons. They are responded to by moving the cursor over the desired button by moving the mouse, and then pressing, or "clicking", the left mouse button.

One item that may appear in a dialog box is a scrollable selection list. This simply contains a list of items the user is able to select from. An item is selected by moving the cursor over the desired item and clicking the left mouse button. The right side of the scollable list will have a vertical bar with a triangular button at each end. If the list contains more items than can be shown at one time, the bar, called a slider, will be shorter than the rest of the list. Clicking the mouse button over either triangular button will scroll the list from that direction to show the additional items. The bar will also move to show the relative position of the list that is being displayed.

B.2.1.3 On-line Help

1. To see what help is available, click the *Help* button on the right side of the menu bar.

- An information dialog box will appear with descriptions of the menus and commands as well as the definitions of the various layers of information that are available for each component.
- Read through the definitions of the layers to see what information is provided for each component.

2. Click on either *OK* or *Cancel* button on the dialog box when done reading the help information to remove the box.
3. NOTE: The help screen is *not* available when another dialog box is on the screen.

B.2.2 Finding and Viewing a Component This section describes the general steps to find a component that performs a specific function and then to view the information on the component.

1. The first step is to indicate what kind of component is desired. This is done by specifying some search criteria. Click on the *Search* button on the menu bar and choose *New Search.*
   - A dialog box should appear to request the component's desired function, the object on which the function acts, the medium in which the function executes, and the domain to which the function belongs. Only the function is required to be selected.
2. Choose the keywords from the selection lists that describe the desired component.
   - Choose "Sort" from the function selection list. It should be highlighted in the top of the function selection box.
   - Leave the other criteria at their default: "(any)".

3. Click on the *OK* button on the dialog box.

    - A new dialog box should appear showing the results of the library search for the given criteria. It also has a selection box showing which information layers can be viewed.

4. Next select which component to view from the component selection list. Also choose the layer to view. The functionality layer is the default.

    - For the Sort example, choose "Bubble_Sort". Leave the default functionality layer.

5. Click the *OK* button on the dialog box.

    - A new window will appear with the specific layer description for the selected component.

6. Close the component information window when done by clicking the *Done* button in the window.

    - Close the Bubble_Sort function view.

B.2.3 Viewing Several Components This section describes how to view several components for comparison.

1. To use the previous search, just click the *View* button on the menu bar to get the dialog box with the results of the previous search. Then get a desired component as described in steps 4-6 in Section B.2.2.

    - From the previous example, the sort functions should be displayed. Select "Bubble_Sort" again, leave the default functionality layer and click on the *OK* button.

2. Next minimize the displayed component window by clicking the close button in the very upper left corner of the component information window. The window icon should appear at the bottom of the screen.

3. Click the *View* menu button again and select another component to view.

   - Perform the steps to view the functionality layer description for the Heap_Sort. Then minimize this window.
   - Finally click the *View* menu button again and bring up the functionality layer for the Merge_Sort component.

4. To compare all selected components, double click on the minimized icons and then move the windows around so that all of them can be viewed at once.

5. Press the *Done* button on the appropriate views when done viewing them.

   - Press the *Done* button on all the components on the screen when done viewing them.

B.2.4 Viewing the Design Layer The design layer has some additional features that are explained in this section.

1. Get a search list that has the desired components to examine.

   - Click the *View* button on the menu bar to bring up the last search list, which should be the sort functions. The component selection dialog box should appear.

2. Select a component and the design layer to view [see steps 4-6 in Section B.2.2 if necessary].

   - Select the Bubble_Sort and Design layer and click *OK*. A window with the Bubble_Sort design should appear.

3. The design layer can present the information various ways: graphically, textually or both. If both methods are available, the graphical method is shown and an additional button is available in the upper right corner of the window labelled "Text...".

   - The Bubble_Sort design layer should have the *Text...* button. Click on it to see the additional text.

4. When the text has been selected, the button changes to "Remove Text" to indicate how to get the graphical representation back.

   - Click on it to get back to the graphical representation.

5. The design layers can be viewed simultaneously for comparison the same way as the other information layers as discussed earlier.

   - Minimize the Bubble_Sort design window. Then get the current search list back (press the *View* button.)
   - Select the Bubble_Sort1 design for viewing and click *OK*.
   - This is the same function as the first Bubble_Sort component, just presented using a different method, called Plan Calculus.

6. When done viewing the design layers, they are closed by clicking on *Done* just like the other layers.

   - Close the Bubble_Sort1 Design view, but leave the Bubble_Sort.

B.2.5 Viewing Other Component Information Layers This section specifies how the other layers of information can be viewed.

1. Starting from a component view that is already displayed, any of the other information layers can be viewed by pressing the appropriately labeled button that are on the right side of the component information window.

   - The Bubble_Sort design layer should still be visible. Press the *Implem.* button on the view to see the implementation layer (source code) for the Bubble_Sort component. A text window should appear.
   - When done, the text window has to be closed like any normal window (pressing the RIGHT mouse button while the cursor is on the title bar and then selecting Quit).
   - Press the *Quality* button to see the view for the Bubble_Sort quality information.
   - Press the *Done* button on all remaining layers when done viewing them.

2. If no component is currently being viewed, click the *View* menu button to bring up the component selection dialog box.
3. Select the desired component and then click on the desired information layer for the component.

   - Select the Quick_Sort Implementation layer .

4. Click the *OK* button to see the information.
5. Close the component windows when done viewing the information.

B.2.6 Narrowing the List of Components One problem that may occur is that the initial list of components may be very long and may include components that are not desired. This can be taken care of by revising the current search criteria to narrow the list of matched components. This is accomplished in the following way:

1. First get the current search criteria for revision. Click on the *Search* button on the menu bar and select *Revise Search.*

   - A dialog box will appear showing the current search criteria.

2. Next, change or specify additional criteria for the desired component. This may include the type of objects of the function, the medium of operation of the function, or the application domain of the function.

   - For the sort example, select "integers" from the object selection list and "array" from the medium selection list.

3. Click on *OK* when the new criteria are as desired.

4. If the new list is still too long, try again.

   - Click on *Cancel* to end this list. Then go back to the Search menu and select *Revise Search* again.

   - This time select "file" as the medium and click on *OK.*

5. When the list reaches a reasonable size, the components can be examined as outlined previously.

   - Simply click on *Cancel* now.

Now that you have completed the instruction manual and followed the examples, you may go back and examine the prototype using various combinations of search criteria to see the various components that are currently available.

Appendix C. *Example SUIT Figures*

The following figures of the prototype interface show the main library functions that can be performed and the resulting information presented to the user. The figures are simple drawings, not screen captures, that have left some details out to prevent the figures from becoming too cluttered. The omitted details include the shading used to create the three dimensional effect of the windows and the highlights to show which button is active. Each figure is accompanied by a short description of what function was executed to generate the results shown. Refer back to Appendix B for explanations of how the functions are executed.

Figure 22 shows what the prototype Reusable Software Library looks like when it is first started.

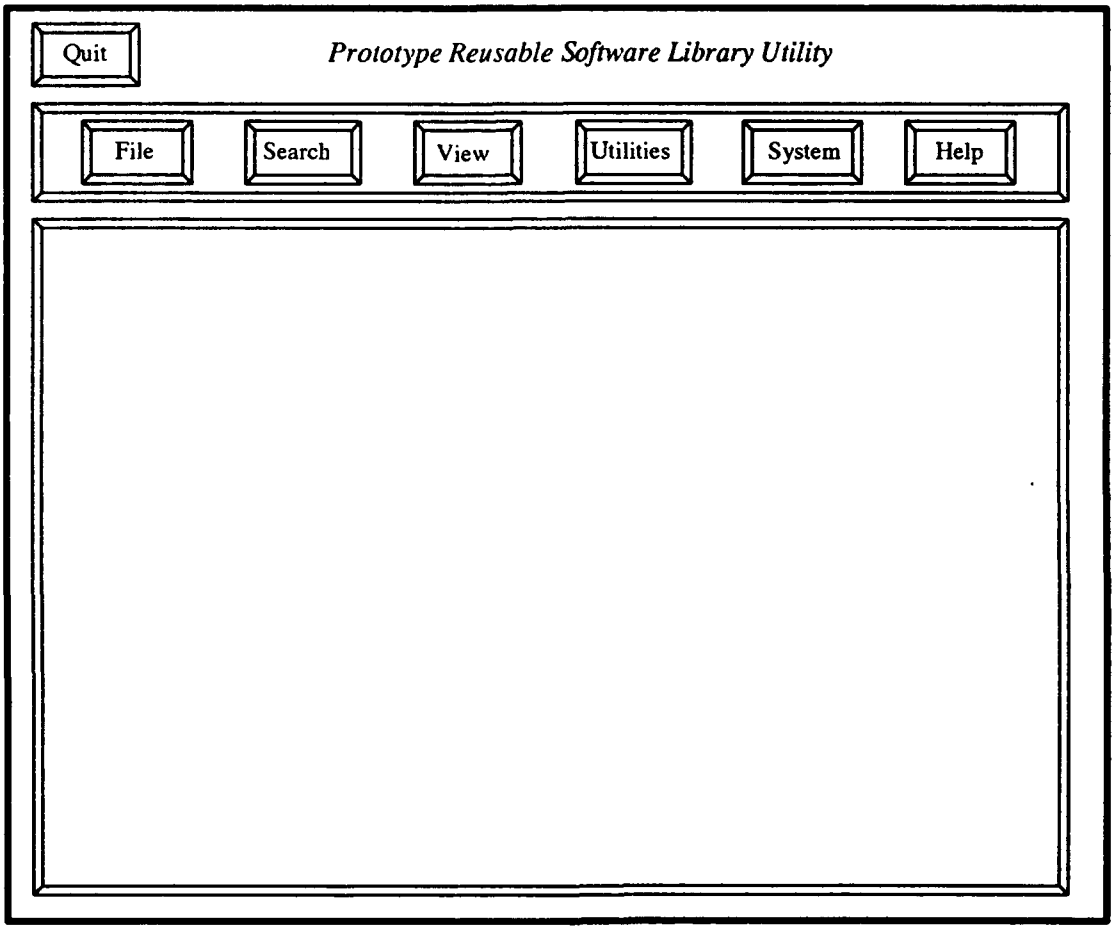

Figure 22. Main Prototype Window

Figure 23 shows the dialog box that is displayed when the user selects "New Search" from the *Search* menu. This dialog queries the user for the search criteria that will be used to find a desired component.

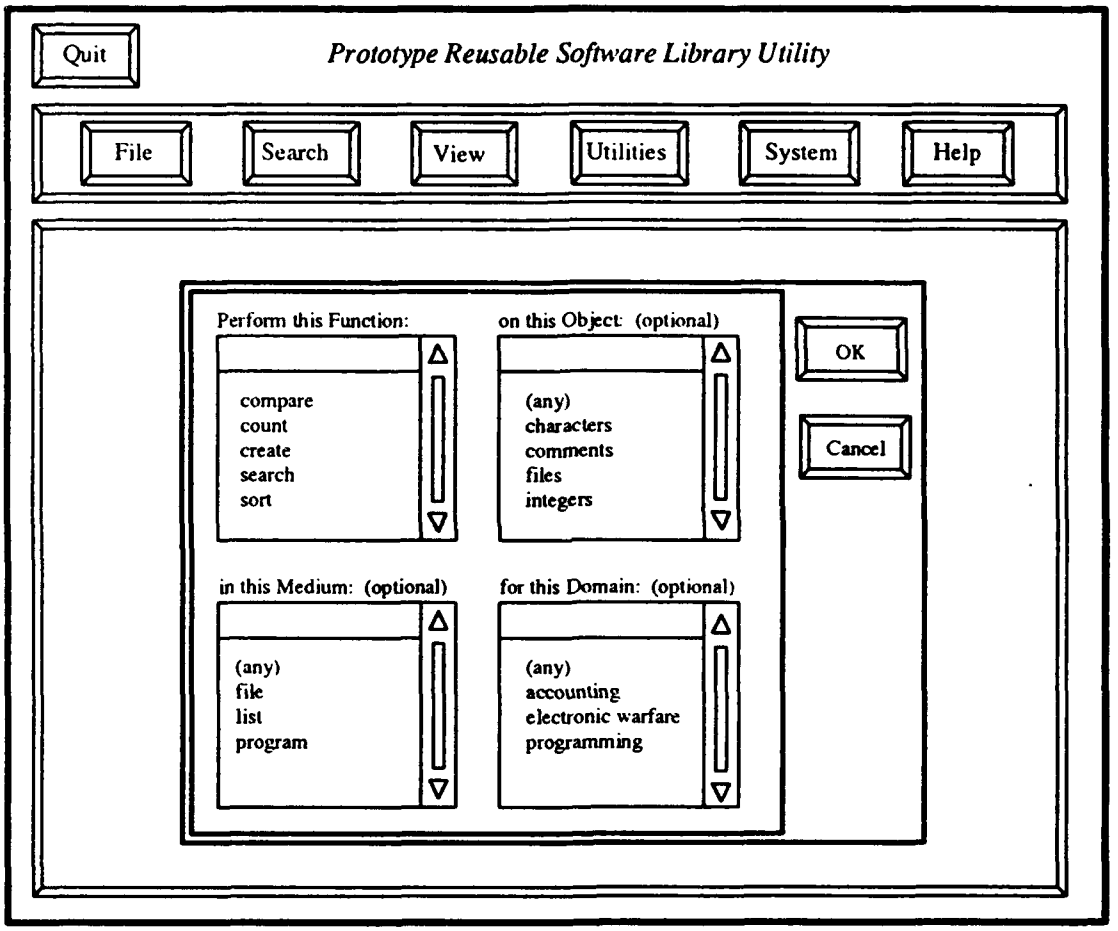

Figure 23. Search Criteria Selection Dialog

Figure 24 shows the dialog box that is displayed after a search has been completed. It shows what key words were used for the search and what components were matched in the database. The user is then able to select a particular component and information layer for viewing.

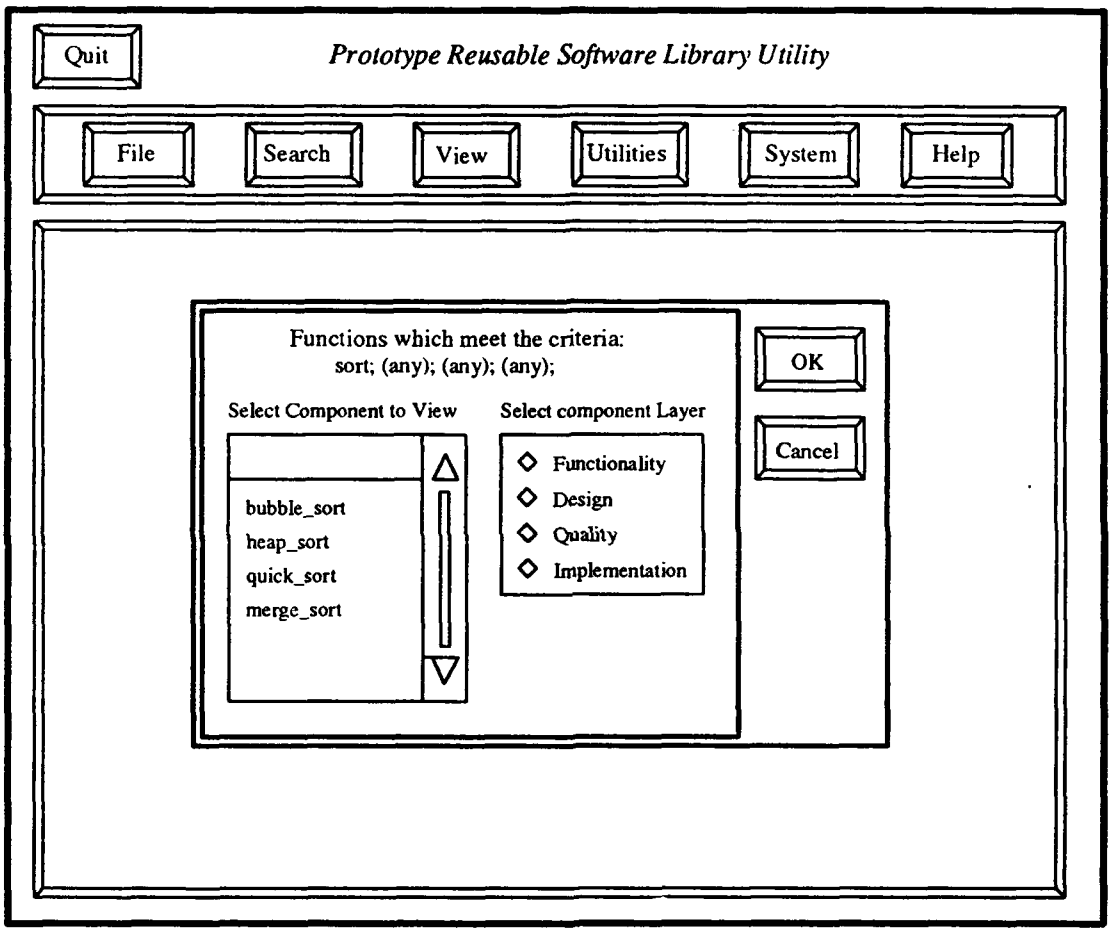

Figure 24. Search Results/Component Selection Dialog

Figure 25 shows the dialog box that is displayed when the user selects "Revise Search" from the *Search* menu. This dialog allows the user to alter the previous search criteria used to find components.

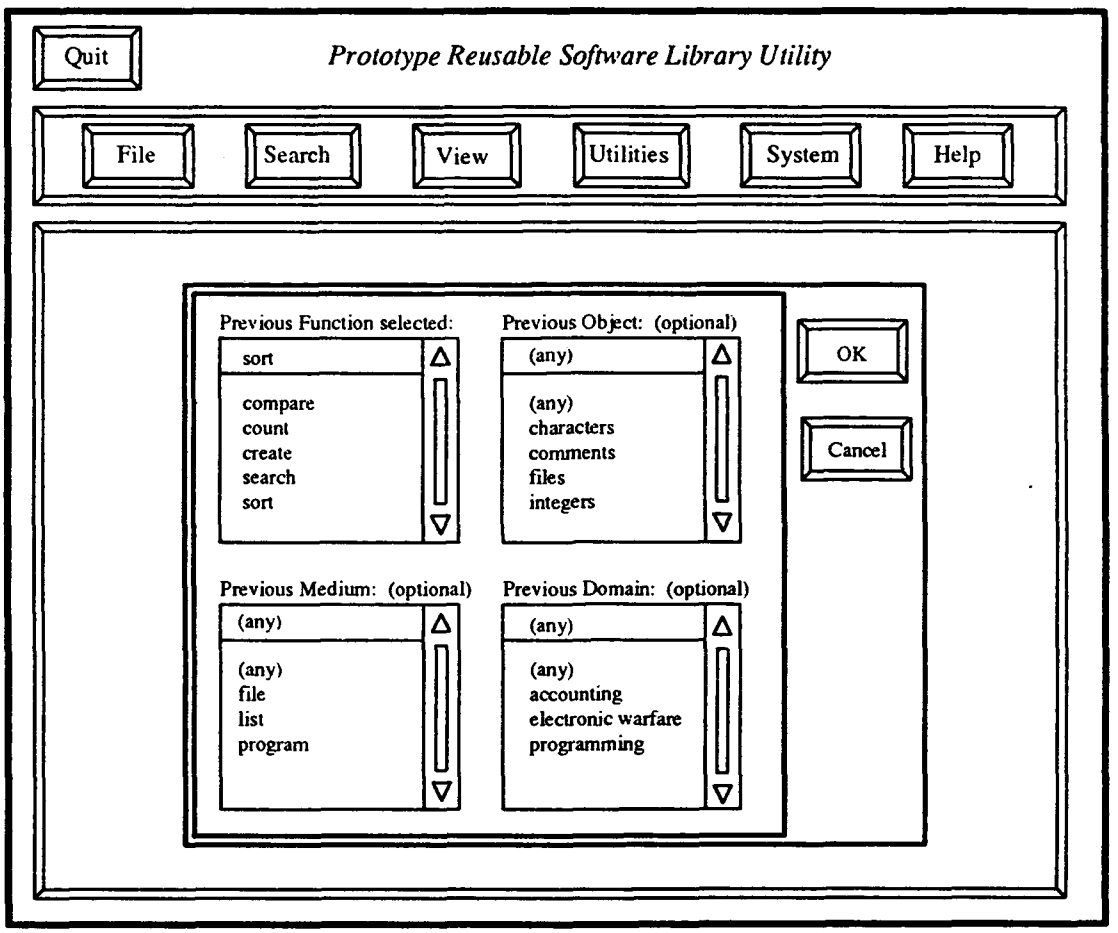

Figure 25. Revise Previous Search Criteria Dialog

Figure 26 shows the window that is presented after the user chooses the functionality layer of a component to view. An example component is shown.

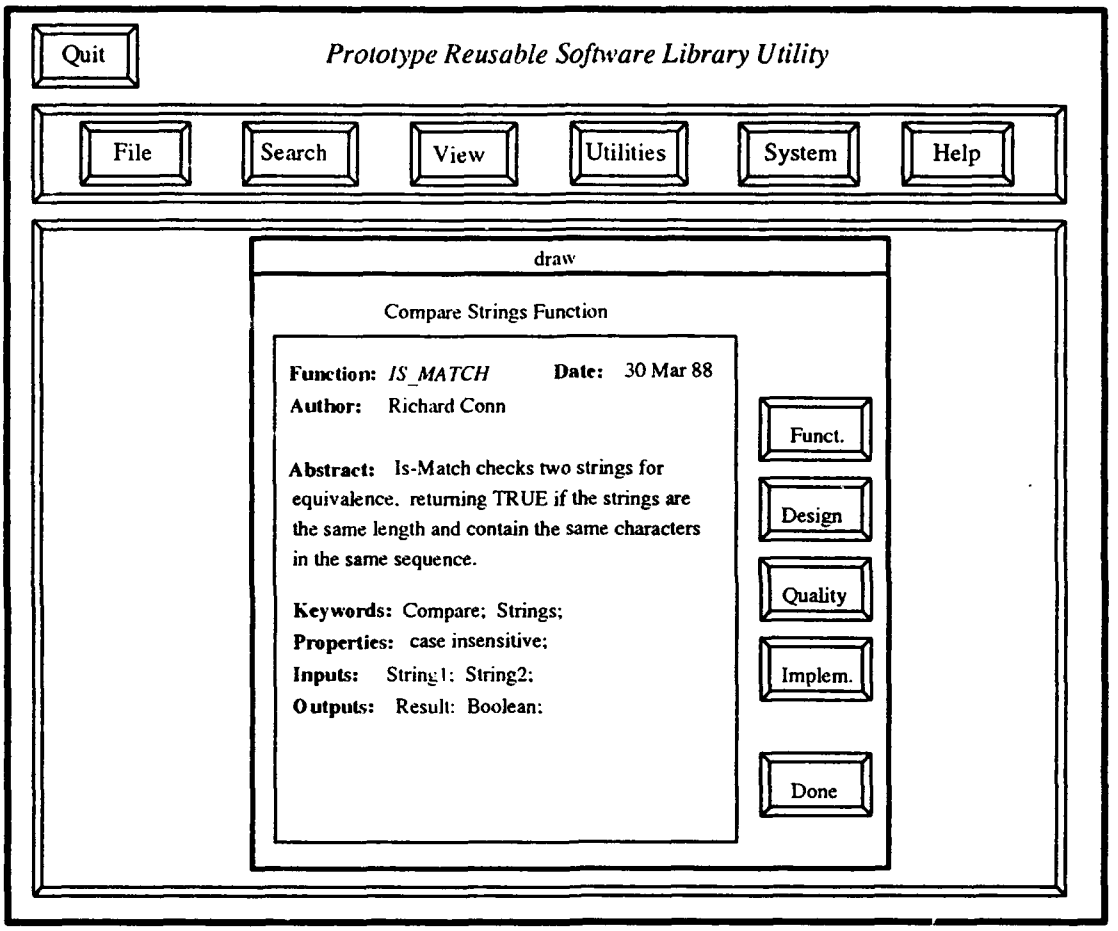

Figure 26. Software Component Functionality Representation

Figure 27 shows the window that is presented after the user chooses the design layer of a component to view. An example component is shown using the PDL design representation. The button in the upper right corner that says "Text..." indicates that a text description is also available.

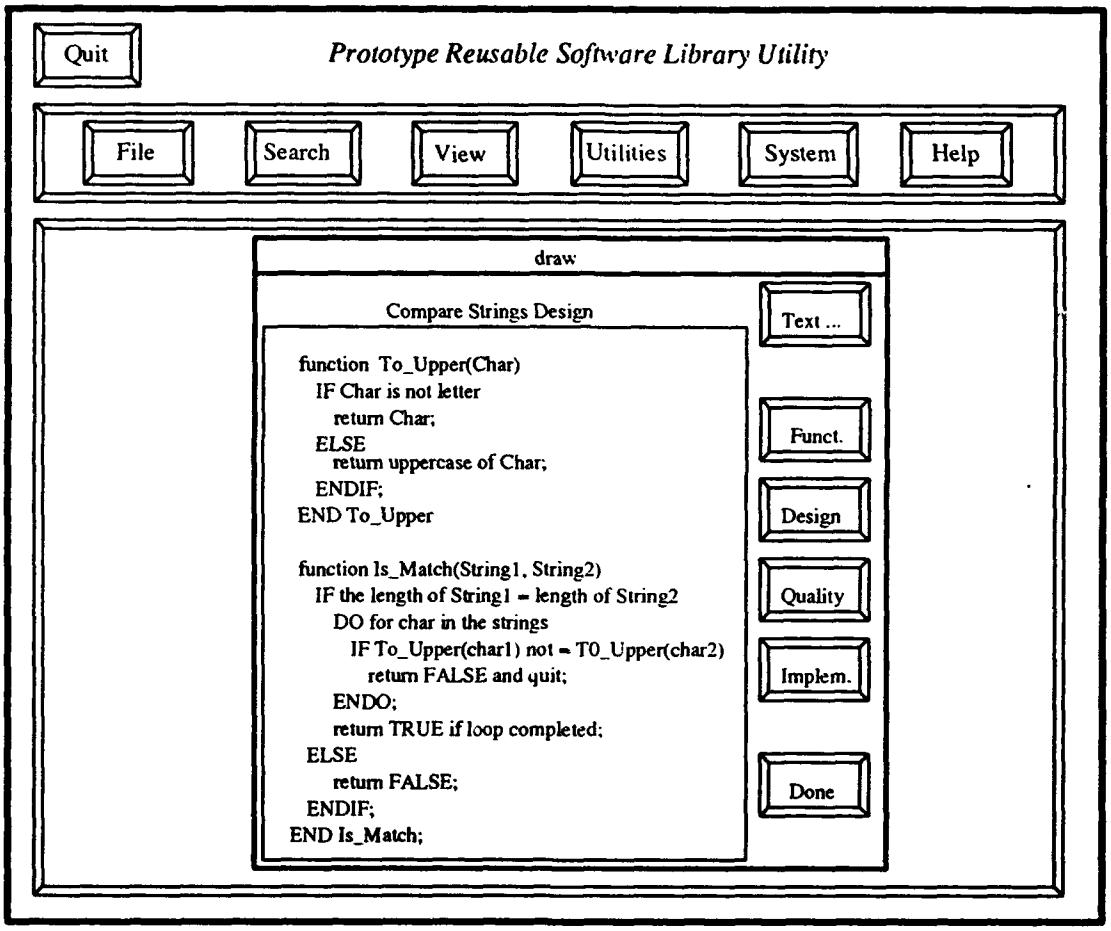

Figure 27. Software Component Design Representation (PDL)

Figure 28 shows the how the design layer representation changes after the user pressed the "Text..." button in the upper right corner to view the associated text description.

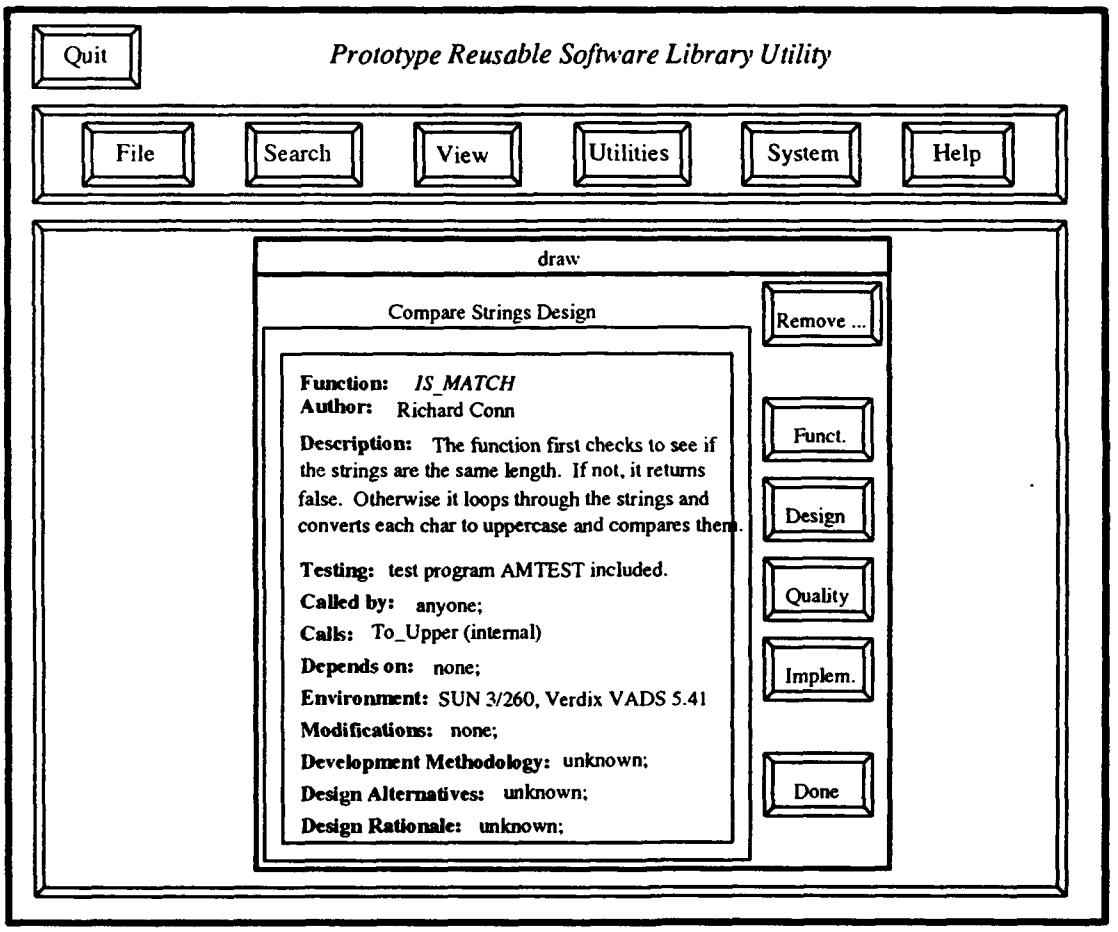

Figure 28. Software Component Design Representation (Text)

Figure 29 shows the type of information that is displayed when the user selects the *Help* button.

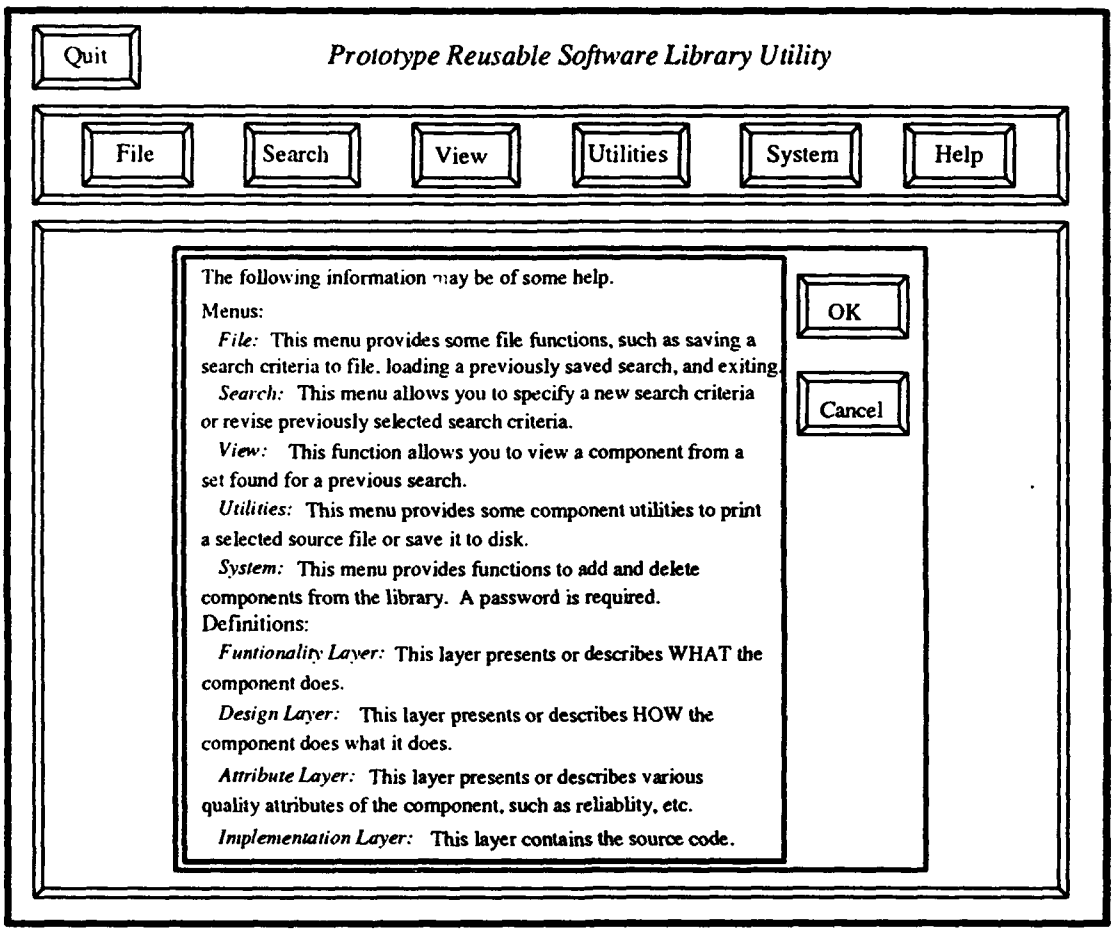

Figure 29. Help Information

Appendix D. *Questionnaire*

The following questionnaire was provided to the prototype evaluators to get feedback on the strengths and weaknesses of the layered approach concept and the representations selected from the evaluation.

Demonstration Questionnaire

I. Background questions:

Answer the following questions, circling the appropriate answer.

1. How much software development experience have you had?
   none 0-3 yrs 4-6 yrs 7-10 yrs 11 yrs +

2. How many lines of software code have you written as part of a commercial or government project?
   none < 1K 1K-10K 11K-50K 51K-100K 101K +

3. How many lines of software code have you written specifically for reuse?
   none < 1K 1K-10K 11K-50K 51K-100K 101K +

4. How many lines of reusable software code have you included in any software development efforts?
   none < 1K 1K-10K 11K-50K 51K-100K 101K +

5. How much time have you spent in the last year looking in software libraries or collections for reusable software?
   none 1-4 hrs 5-10 hrs 11-20 hrs 21-50 hrs+ 51 hrs+

6. What software engineering education/training have you had (check all that apply)?
   - ☐ on-the-job training
   - ☐ college courses
   - ☐ college degree
   - ☐ graduate courses
   - ☐ graduate degree
   - ☐ professional continuing education

II. *Functionality Layer*

The following set of statements are all related to the component functionality descriptions presented by the prototype. Browse through the prototype library and examine the component functionality descriptions. Then write the number that indicates how much you agree with the following statements on the line provided using the following scale:

strongly disagree	somewhat disagree	borderline	somewhat agree	strongly agree
1	2	3	4	5

1. _____ The component functionality representation is independent of specific programming languages.

2. _____ The representation is independent of specific development methodologies.

3. _____ The representation is independent of specific applications.

4. _____ The representation specifies what the component does.

5. _____ The representation specifies non-functional requirements.

6. _____ The representation is understandable.

7. _____ The representation is unambiguous.

8. _____ The representation distinguishes between similar items.

9. What type of training or education would be required to understand the component functionality representations? (Circle the appropriate answer)

   (a) High school classes.

   (b) College classes in software development or software engineering.

   (c) A college degree in software engineering

   (d) General work experience in software development.

   (e) Specific training for this particular representation.

   (f) Other: _____

10. What additional information would make the description of the function more complete, understandable, or better able to distinguish between components?

III. *Program Design Language*

The following set of statements are directed at the designs represented using the Programming Design Language (PDL), without considering the associated text description of the design. Browse through the prototype library and examine the PDL design representations. Then write the number of the response that indicates how much you agree with each of the following statements on the line provided using this scale:

strongly disagree	somewhat disagree	borderline	somewhat agree	strongly agree
1	2	3	4	5

11. _____ The PDL design representation is independent of specific programming languages.

12. _____ The representation is independent of specific development methodologies.

13. _____ The representation is independent of specific applications.

14. _____ The representation is understandable.

15. _____ The representation is unambiguous.

16. _____ The representation distinguishes between similar items.

17. _____ The representation specifies the architectural design.

18. _____ The representation specifies system dependancies.

19. _____ The representation specifies design alternatives.

20. _____ The representation specifies design rationale.

21. _____ The representation specifies the design methodology used for development.

22. What type of training or education would be required to understand the PDL design representations? (Circle the appropriate answer)

    (a) High school classes.

    (b) College classes in software development or software engineering.

    (c) A college degree in software engineering

    (d) General work experience in software development.

    (e) Specific training for this particular representation.

    (f) Other: _____

Now consider the text descriptions that are included with the PDL design information and respond to the following statements using the scale:

strongly disagree	somewhat disagree	borderline	somewhat agree	strongly agree
1	2	3	4	5

23. _____ The text description is necessary to understand the component.

24. _____ The text is necessary to distinguish between similar components.

25. _____ The text is necessary to represent the architectural design.

26. _____ The text is necessary to represent the system dependancies.

27. _____ The text is necessary to represent the design alternatives.

28. _____ The text is necessary to represent the design rationale.

29. _____ The text is necessary to represent the design methodology.

30. List any other comments about the PDL design representations:

IV. *Plan Calculus*

The following set of statements are specifically related to the designs represented using the Plan Calculus, without considering the associated text description of the design. Browse through the prototype library and examine the Plan Calculus design represenations. Then write the number of the response that indicates how much you agree with each of the following statements on the line provided using the scale:

strongly disagree	somewhat disagree	borderline	somewhat agree	strongly agree
1	2	3	4	5

31. _____ The Plan Calculus design representation is independent of specific programming languages.

32. _____ The representation is independent of specific development methodologies.

33. _____ The representation is independent of specific applications.

34. _____ The representation is understandable.

35. _____ The representation is unambiguous.

36. _____ The representation distinguishes between similar items.

37. _____ The representation specifies the architectural design.

38. _____ The representation specifies system dependancies.

39. _____ The representation specifies design alternatives.

40. _____ The representation specifies design rationale for the decisions made.

41. _____ The representation specifies the design methodology used for development.

42. What type of training or education would be required to understand the Plan Calculus design representations? (Circle the appropriate answer)

    (a) High school classes.

    (b) College classes in software development or software engineering.

    (c) A college degree in software engineering

    (d) General work experience in software development.

    (e) Specific training for this particular representation.

    (f) Other: _____

Now consider the text descriptions that were included with the Plan Calculus design information and respond to the following statements using the scale:

strongly disagree	somewhat disagree	borderline	somewhat agree	strongly agree
1	2	3	4	5

43. _____ The text description is necessary to understand the component.

44. _____ The text is necessary to distinguish between similar components.

45. _____ The text is necessary to represent the architectural design.

46. _____ The text is necessary to represent the system dependancies.

47. _____ The text is necessary to represent the design alternatives.

48. _____ The text is necessary to represent the design rationale.

49. _____ The text is necessary to represent the design methodology.

50. List any other comments about the Plan Calculus design representations:

51. What additional information would make the design descriptions more complete, understandable, or better able to distinguish between components?

52. What other comments do you have about the design layer that have not been addressed in this survey?

V. *Prototype Questions*

Write in the number of the response that indicates how much you agree with the given statements on the line provided using the following scale:

strongly disagree	somewhat disagree	borderline	somewhat agree	strongly agree
1	2	3	4	5

53. _____ Presenting the components in layers is helpful for understanding.

54. _____ Presenting the components in layers is helpful for comparing components.

55. _____ The four layers cover component information well.

56. _____ The interface is easy to use.

57. _____ The menus are arranged in a logical manner.

58. What other comments do you have about the prototype that have not been addressed in this survey?

Appendix E. *Prototype Questionnaire Responses*

E.1 Background Questions

Question/ Choices	Respondent 1	2	3	4	5	6	7
Ques. 1: Software development experience							
none							X
0-3 yrs	X		X	X		X	
4-6 yrs							
7-10 yrs					X		
11 yrs +		X					
Ques. 2: SLOC written							
none			X	X			X
< 1K							
1K-10K	X					X	
11K-50K		X			X		
51K-100K							
101K +							
Ques. 3: Reusable SLOC written							
none	X		X				
< 1K		X		X			
1K-10K					X	X	X
11K-50K							
51K-100K							
101K +							
Ques. 4: Reusable SLOC used							
none			X		X		X
< 1K		X		X			
1K-10K	X					X	
11K-50K							
51K-100K							
101K +							
Ques. 5: Time browsing reusable libraries							
none	X						
1-4 hrs		X		X	X	X	
5-10 hrs			X				X
11-20 hrs							
21-50 hrs							
51 hrs +							

E.2 Prototype Questions

Question No.	Respondent 1	2	3	4	5	6	7	Avg	Std Dev
Functionality Layer									
1	4	5	5	5	5	5	5	4.86	0.3499
2	4	5	5	5	5	5	5	4.86	0.3499
3	5	4	5	5	5	5	5	4.86	0.3499
4	4	5	5	4	4	5	4	4.43	0.4949
5	3	3	5		5	2	5	3.83	1.2134
6	5	5	5	4	5	5	5	4.86	0.3499
7	5	4	3	4	5	4	5	4.29	0.6999
8	5	2	1	3	4	4	5	3.43	1.3997
9	b	d	b	d	d	b	b	n/a	n/a
PDL Design without Text									
11	4	4	4	5	4	5	4	4.29	0.4518
12	4	4	4	5	5	5	5	4.57	0.4949
13	5	5	4	5	5	4	5	4.71	0.4518
14	4	5	5	4	5	5	5	4.71	0.4518
15	4	3	5	4	5	4	5	4.29	0.6999
16	5	4	5	5	5	4	5	4.71	0.4518
17	4	3	4	5	5	5	5	4.43	0.7284
18	4	2	4	1	3	1	4	2.71	1.2778
19	2	2	4	1	2	1	5	2.43	1.3997
20	3	1	4	1	2	1	5	2.43	1.4983
21	3	2	4	2	3	1	5	2.86	1.2454
22	d	d	b	d	d	a	b	n/a	n/a
PDL Design with Text									
23	3	3	3	4	4	4	4	3.57	0.4949
24	3	2	2	5	5	5	3	3.57	1.2936
25	3	1	3	2	2	1	2	2.00	0.7559
26	4	2	3	5	5	5	5	4.14	1.1249
27	4	2	4	5	5	2	4	3.71	1.1606
28	4	4	4	5	5	4	5	4.43	0.4949
29	4	2	4	4	4	4	5	3.86	0.8330

Question No.	Respondent 1	2	3	4	5	6	7	Avg	Std Dev
Plan Calculus Design without Text									
31	4	5	4	5	5	5	5	4.71	0.4518
32	4	5	4	5	4	4	5	4.43	0.4949
33	5	5	4	5	5	4	5	4.71	0.4518
34	2	2	3	3	2	1	3	2.29	0.6999
35	4		4	4	2	1	4	3.17	1.2134
36	3	2	4	4	2	5	5	3.57	1.1780
37	3	3	4	4	3	4	4	3.57	0.4949
38	3	1	3	1	1	1	5	2.14	1.4569
39	3	3	3	1	1	1	5	2.43	1.3997
40	3	1	3	1	1	1	5	2.14	1.4569
41	4	2	3	1	2	1	5	2.57	1.3997
42	e	e	b	e	e	e	e	n/a	n/a
Plan Calculus Design with Text									
43	5	5	4	5	5	5	3	4.57	0.7284
44	4	4	3	4	4	3	4	3.71	0.4518
45	4	2	3	3	3	3	3	3.00	0.5345
46	4	4	4	5	5	5	5	4.57	0.4949
47	4	4	3	5	5	2	2	3.57	1.1780
48	4	3	3	5	5	4	5	4.14	0.8330
49	4	2	3	4	3	4	5	3.57	0.9035
Prototype Questions									
53	5	5	5	5	5	5	5	5.00	0.0000
54	5	5	5	4	5	5	5	4.86	0.3499
55	4	4	5	4	5	5	5	4.57	0.4949
56	5	5	5	5	5	5	5	5.00	0.0000
57	5	5	5	5	5	5	5	5.00	0.0000

NOTE: The following questions were not answered by the indicated respondents. The missing values did not affect the calculations for the average and standard deviation for each question.

Question	Respondent
5	4
35	2

Bibliography

1. Arango, Guillermo and others. "A Tool Shell for Tracking Design Decisions," *IEEE Software*, *8*:75–83 (March 1991).
2. Arango, Guillermo and Ruben Prieto-Diaz. "Domain Analysis Concepts and Research Directions." *Domain Analysis and Software Systems Modeling* edited by Ruben Prieto-Diaz and Guillermo Arango, Los Alamitos CA: IEEE Computer Society Press, 1991.
3. Biggerstaff, Ted and Charles Richter. "Reusability Framework, Assessment, and Directions," *IEEE Software*, *4*:3–11 (March 1987).
4. Booch, Grady. *Software Components with Ada*. Menlo Park CA: Benjamin/Cummings Publishing Co., 1987.
5. Bowen, Thomas P. and others. *Specification of Software Quality Attributes*. Final Technical Report RADC-TR-85-37, Vol. I, Griffiss AFB NY: Rome Air Development Center, February 1985 (AD-A153 988).
6. Brooks, Fred P. "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, *20*:10–19 (April 1987).
7. Burton, Bruce A. and others. "The Reusable Software Library," *IEEE Software*, *4*:25–33 (July 1987).
8. Caine, Stephen H. and E. Kent Gordon. "PDL – A Tool for Software Design." *Tutorial on Software Design Techniques, 4th Ed.* edited by Peter Freeman and Anthony I. Wasserman, Los Alamitos CA: IEEE Computer Society Press, 1983.
9. Caldiera, Gianluiga and Victor R. Basili. "Identifying and Qualifying Reusable Software Components," *Computer*, *24*:61–70 (February 1991).
10. Cardow, Capt James and Capt William Watson. Class handout distributed in WCSE 474, Software Generation and Maintenance; Library Management and Construction section. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1991.
11. Cardow, Capt James, Instructor, WSCE 474, Software Generation and Maintenance. Personal interviews. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 30 March through 13 September 1992.
12. Comer, Edward R. "Domain Analysis: A Systems Approach to Software Reuse." *9th Digital Avionics Systems Conference*. 224–229. Piscataway NJ: IEEE Service Center, 1990.
13. Conklin, Jeff. "Hypertext: An Introduction and Survey," *IEEE Computer*, *20*:17–40 (September 1987).

14. Conklin, Jeff. "Design Rationale and Maintainability." *Proceedings of the 22th Hawaii International Conference on System Sciences, Vol. 2*. 533-539. North Hollywood CA: Western Periodicals Co., 1989.

15. Devanbu, Premkumar and others. "LaSSIE, a Knowledge-based Software Information System," *Communications of the ACM*, *34*:34-49 (May 1991).

16. Dubois, Eric and others. "A knowledge Representation Language for Requirements Engineering." *Proceedings of the IEEE*. 1431-1443. Piscataway NJ: IEEE Service Center, 1986.

17. Frakes, W. B. and P. B. Gandel. "Representation Methods for Software Reuse." *Tri-Ada 89 Proceedings*. 302-314. New York: ACM Press, 1989.

18. Frakes, W. B. and P.B. Gandel. "Classification, Storage, and Retrieval of Reusable Components." *Proceedings of 12th International ACMSIGIR Conference on Research and Development in Information Retrieval*. 251-254. New York: ACM Press, 1989.

19. Frakes, W. B. and B. A. Nejmeh. "An Information System for Software Reuse." *IEEE Tutorial on Software Reuse: Emerging Technology* edited by W. Tracz, Los Alamitos CA: IEEE Computer Society, 1988.

20. Franke, David W. "Deriving and Using Descriptions of Purpose," *IEEE Expert*, *6*:41-47 (April 1991).

21. Gane, C. P. "Data Design in Structured Systems Analysis." *Tutorial on Software Design Techniques, 4th Ed.* edited by Peter Freeman and Anthony I. Wasserman, Los Alamitos CA: IEEE Computer Society Press, 1983.

22. Ince, Darrel. "Z and System Specification," *Information and Software Technology*, *30*:138-145 (April 1988).

23. Jordan, Kathleen A. and Alan M. Davis. "Requirements Engineering Metamodel: An Integrated View of Requirements." *IEEE 15th Annual International Computer Software and Applications Conference (COMPSAC91)*. 472-478. Los Alamitos CA: IEEE Computer Society Press, 1991.

24. Karat, John. "Software Evaluation Methodologies." *Handbook of Human-Computer Interaction* edited by Martin Helander, New York: Elsevier Science Publishing Co., Inc., 1988.

25. Lubars, Mitchell D. "Domain Analysis and Domain Engineering in IDeA." *Domain Analysis and Software Systems Modeling* edited by Ruben Prieto-Diaz and Guillermo Arango, Los Alamitos CA: IEEE Computer Society Press, 1991.

26. Lubars, Mitchell D. and Mehdi T. Harandi. "Addressing Software Reuse Through Knowledge-Based Design." *Software Reusability, Vol. II: Applications and Experience* edited by Ted J. Biggerstaff and Alan J. Perlis, New York: ACM Press, 1989.

27. Matsumoto, Yoshihiro. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels." *Software Reusability, Vol. II: Applications and Experience* edited by Ted J. Biggerstaff and Alan J. Perlis. New York: ACM Press, 1989.

28. McCall, Jim A. and others. *Factors in Software Quality: Concept and Definitions of Software Quality*. Final Technical Report RADC-TR-77-369, Vol. I, Griffiss AFB NY: Rome Air Development Center, November 1977 (AD-A04 9014).

29. Oman, Paul and Jack Hagemeister. *Metrics for Assessing a Software System's Maintainability*. Report 92-01-TR, University of Idaho: Software Engineering Test Lab, March 1992.

30. Prieto-Diaz, Ruben. "Implementing Faceted Classification for Software Reuse." *Communications of the ACM*, *34*:89-97 (May 1991).

31. Prieto-Diaz, Ruben and Peter Freeman. "Classifying Software for Reusability." *IEEE Software*, *4*:6-16 (January 1987).

32. Privitera, Dr. J. P. "Ada Design Language for the Structured Design Methodology." *Tutorial on Software Design Techniques, 4th Ed.* edited by Peter Freeman and Anthony I. Wasserman, Los Alamitos CA: IEEE Computer Society Press, 1983.

33. Rich, Charles. "A Formal Representation for Plans in the Programmer's Apprentice." *Proceedings of the 7th International Joint Conference on Artificial Intelligence*. 1044-1052. Los Altos CA: International Joint Conference on Artificial Intelligence, 1981.

34. Rich, Charles and Richard C. Waters. "Formalizing Reusable Software Components in the Programmer's Apprentice." *Software Reusability, Vol. II: Applications and Experience* edited by Ted J. Biggerstaff and Alan J. Perlis, New York: ACM Press, 1989.

35. Webster, Dallas E. *Mapping the Design Information Terrain*. Technical Report STP-367-88, Austin TX: Microelectronics and Computer Technology Corporation, November 1988.

36. Worrall, Capt Gary G. *A Hypermedia Implementation for Reusable Software Component Representation*. MS thesis, AFIT/GCS/ENG/90D-16, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A230497).

37. Yourdon, Edward. *Modern Structured Analysis*. Englewood Cliffs NJ: Yourdon Press, 1989.

Vita

Captain Paul D. Siebels was born on 18 April 1965 in Newton, Iowa. He graduated as Valedictorian from Newton Senior High School in 1983. He attended Rose-Hulman Institute of Technology in Terre Haute, Indiana with an Air Force ROTC scholarship. He graduated *magna cum laude* with a Bachelor of Science degree in Electrical Engineering in May 1987 and received a reserve commission in the US Air Force. His first assignment was to Wright-Patterson AFB, Ohio as an Avionics Systems Engineer for the Deputy of Engineering, Aeronautical Systems Division, Air Force Systems Command. He managed and recommended technical improvements for the computer software for the YA-7F and C-29A programs. In February 1989 he was assigned to the Joint Tactical Autonomous Weapons (JTAW) System Program Office (SPO) as a Computer Resource Engineer. He managed the technical development of the Mission Computer hardware and software for the Tacit Rainbow program until entering the School of Engineering, Air Force Institute of Technology, in May 1991.

Permanent address: 71 Hidden Cove Lane
Valparaiso, Florida 32580

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Dec 92	Master's Thesis

4. TITLE AND SUBTITLE
Examining a Layered Approach to Function and Design Representation for Reusable Software Components

5. FUNDING NUMBERS

6. AUTHOR(S)
Paul D. Siebels, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology,
WPAFB, OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/92D-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Software Technology for Adaptable, Reliable Systems (STARS)
Suite 400 801 N. Randolph Street
Arlington VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This effort examined ways to improve the effectiveness of reusable software libraries. The main area of investigation was in improving the user interface by finding better ways to present the software components to potential re-users. The first aspect which was considered was finding an effective representation for reusable software components. A set of criteria was developed for evaluating the effectiveness of software representations. The criteria consisted of generality, expressiveness, understandability, consistency, and resolution. The second aspect which was considered was how to present the software component information to the user to facilitate finding the appropriate component for reuse. A representation framework was examined which advocated presenting reuse information in four layers: component functionality, design information, quality metrics, and source code. Several current representations for software function and design were evaluated using the criteria listed above. The highest rated representations were then incorporated into a prototype library interface for examination by a group of software engineers. Feedback was collected and summarized in a set of recomendations and conclusions.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software reuse; Computer programs; Libraries; Reusable equipment; Software engineering; Information retrieval;	114
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL