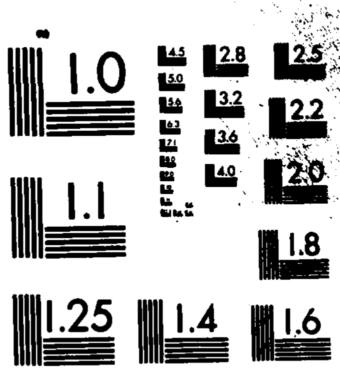
AD-A159 386 COGNITIVE ISSUES IN SOFTWARE REUSE(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA E M COELHO JUN 85 1/1

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A159 386

THESIS

COGNITIVE ISSUES
IN
SOFTWARE REUSE

by

Eduardo M. P. Coelho

June 1985

Thesis Advisor: Gordon H. Bradley

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A159 386	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Cognitive Issues in Software Reuse		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; June 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Eduardo M. P. Coelho		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100		12. REPORT DATE June 1985
		13. NUMBER OF PAGES 75
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

knowledge domain, memory, short-term-memory, long-term-memory, software reusability, DRACO Paradigm

20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Vast resources are invested in the construction of software. Reuse of software offers potential savings in the construction of new software systems. From the perspective of cognitive science, current proposals for software reuse are depicted. This work starts with a cognitive analysis of programming behavior (human thought processes). The aspects of cognitive behavior related to program comprehension, the notions of knowledge domain, knowledge acquisition and reconstruction and memory mechanisms are discussed. The definition of software reusability is presented and methods to achieve reuse are discussed. The software development model called DRACO is presented and its concepts are related to software reuse and reconstruction.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Approved for public release; distribution is unlimited.

Cognitive Issues in Software Reuse

by

Eduardo M. P. Coelho
Lieutenant Comander of Portuguese Navy
B.S., Portuguese Naval Academy, 1968.

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

Author: ______________________
Eduardo M. P. Coelho

Approved by: ______________________
Gordon H. Bradley, Thesis Advisor

____
Bruce J. MacLennan, Second Reader

____
Bruce J. MacLennan, Chairman,
Department of Computer Science

____
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

Vast resources are invested in the construction of software. Reuse of software offers potential savings in the construction of new software systems. From the perspective of cognitive science, current proposals for software reuse are depicted. This work starts with a cognitive analysis of programming behavior (human thought processes). The aspects of cognitive behavior related to program comprehension, the notions of knowledge domain, knowledge acquisition and reconstruction and memory mechanisms are discussed. The definition of software reusability is presented and methods to achieve reuse are discussed. The software development model called DRACO is presented and its concepts are related to software reuse and reconstruction.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. THE SOFTWARE CRISIS

In the last few years more than fifty billions of dollars was spent on software production and maintenance in the United States[Ref. 1]. This enormous sum was spent on something which cannot be seen or touched in the conventional sense. The specific nature of software has brought on many of the problems in its production. In the last years the problem of software production has been growing rapidly with the increased size of the software systems. In the near future "personal computers" will be able to hold the largest software systems built. Unless techniques to create software dramatically increase in productivity, we will not be able to effectively use this enormous increase in computer power.

Because of this we can use the term "software crisis" meaning that there is a demand for quality of software which cannot be met with present methods of software construction. Some of the points which have caused the software crisis are listed below:

The price/performance of computing hardware has been decreasing (about 20% per year)[Ref. 2];

The total installed processing capacity is increasing (about 40% per year)[Ref. 2];

As computers become less expensive they are used in more applications areas, all of which demand software;

The cost of software as a percentage cost of a total computing systems has been increasing[Ref. 3];

The productivity of the software creation process has increased only 3% - 8% per year for the last twenty years[Ref. 2];

As the size of the software system grows, it becomes increasingly hard to construct;

There is a shortage of qualified personnel to create software[Ref. 4].

B. THE SOFTWARE LIFECYCLE

The beginning of the software crisis was announced by the failure of some very large software systems to meet their analysis goals and delivery dates in the 1960's. These systems failed in spite of the amount of money and manpower allocated to the projects. These failures originated an analysis of the problems of software construction which marked the beginning of software engineering.

Several studies of the process of software construction have identified the phases that a software project goes through and these phases have been combined into a model called the software lifecycle[Refs. 3,5]. If we view the lifetime of a software system as consisting of the phases: requirements analysis, design, code and testing, and maintenance then the average cost associated with the phases are[Ref. 3]:

- Requirements analysis..........9%
- Design 6%
- Code and testing 15%
- Maintenance 70%

If a tool is developed to help the production of software its impact depends on the importance of the lifecycle phases it affects. Thus a design tool has the least impact while the maintenance tool has potentially the most impact.

C. REUSABILITY AND COGNITIVE SCIENCES

One attempt to reduce software costs has focused on incorporating software products produced in previous projects into projects that are under development. This approach is called "software reusability" and it involves trying to incorporate whole or partial software products such as code, analysis plans, requirements design, test plans, etc. Software reuse has been an active research area and there has been considerable discussion about the obvious economic benefits. But despite the considerable interest, there has been very little actual reuse of software products.

The current enthusiasm for reusability seems to be based on the assumption that if software exists that performs the same (or nearly the same) function as the product under development, it should be found and used. This assumption represents a simple and very naive view of the programmer's role in software development process. Recent work on cognitive sciences has lead to the development of some more sophisticated (and hopely more accurate) views of the programming process. Here this work on cognitive science is reviewed and then, from this perspective, current proposals for software reuse are analysed.

The section of the thesis on cognitive models depicts the memory mechanism, the knowledge involved in the components of the memory and the techniques to increase memory capacity (chunking). The cognitive aspect in computer programming, which includes the concepts of domains, its application to reusability and the issue of "documentation" included in the generic field of external memory, is discussed.

Finally the fundamental idea of this work, software reusability, is presented. The principles of reusability will be discussed and one model,the "DRACO PARADIGM" based on reusable principles will be presented. Using this model we analyze how to create software and the way its maintenance and design recovery is acomplished.

II. COGNITIVE SCIENCE IN SOFTWARE ENGINEERING

A. INTRODUCTION

More and more in the study of programming and programming languages, human factors directly related with the behavior of the programmers and the human mind itself become important. How we think, our limitations and capabilities play a fundamental role in the organization of the human thought process. The thinking process is based on the understandabilty of a stimulus, how it affects us and the way in which the information of a stimulus is processed. In programming the stimulus can be code, design, software tools, or other forms of software information needed to construct and develop a program.

Another issue to consider is the proper cognitive psychology of the human being, that consists of how people perceive, organize, process and remember information. This important mechanism is analysed in the next chapter.

B. COGNITIVE SCIENCE

There exist several theories or approaches to understanding how programmers develop programs. They are usually based on the psychological principles related to memory mechanisms.

Usually the approaches begin with the distinction between short and long-term-memory, its capacity and way it works. Also the concept of "chunking", that expands the capacity of our short-term-memory, is important.

Another important approach is presented by Shneidermann and Mayer[Ref. 6]. They present a model of knowledge based on a syntactic/semantic model and the concept of knowledge domain.

The fundamental idea is related to the acquisition and development of programming skills and consists of the integration of knowledge from several different knowledge domains.

Another model is given by Atwood[Ref. 7] for the comprehension of a program. In his theory he breaks a program into a hierarchical tree structure of statements. After understanding the elementary statements at the bottom of the tree, they are fused into macro statements until the top of the tree is reached. Once this stage is achieved the programmer understands the program. This process is very close to "chunking".

Cognitive science shows one way of representation and organization of the programmer's knowledge and permits one opportunity to control the largest source of influence of project performance.

C. PROGRAM COMPREHENSION

The program comprehension task is a very important one in programming because it is common to several task such as debugging, testing and modification. In program comprehension, programmers have to develop an internal semantic structure for representing the syntax of the program. It is acquired as high level knowledge, so the programmer doesn't need to memorize the program's line-by-line form based on syntax. With the knowledge of internal structure it is possible to do a large variety of transformations on the program like, for instance, converting it to another programming language or developing new data representations.

D. PROBLEM SOLVING MODEL

Problem solving is characterized by a process that develops several steps in a defined order . The first step in this model will be to join and to organize all the material relevant for the problem. Then the problem is fractionated and the data is analyzed to propose solutions for the parts of the problem[Ref. 8] After the several solutions have been analyzed using a process of synthesis, the final solution of the problem is constructed. Finally, the last step consists of the test and verification of the solution.

E. SOFTWARE ENGINEERING KNOWLEDGE

A software development model for the explicit representation and manipulation of domain specific and software engineering knowledge allows us to take a new view of the problem of system evolution and maintenance. The description of a system includes its initial statement, specifications, the software engineering knowledge, the constraints of the generation process, and construction planning heuristics base which encapsulate the design rationalizations and engineering knowledge involved in its current implementation. As a software system evolves due to changes in the content specification, in the software engineering specification or in the operating environment, we can relate these changes to precisely defined portions of the system's descriptions. Either the initial specification can be modified and an executable representation rederived or appropriate manipulation of the system's associated engineering knowledge bases may guide software engineering knowledge in the derivation of alternatives implementations.

III. KNOWLEDGE ACQUISITION AND REPRESENTATION

A. INTRODUCTION

One important component of the human knowledge mechanism is memory which is at once remarkable for its power and for its limitations. On the one hand the vast store of information that we have in memory for the meaning of words, facts and images is considerably superior to the most powerful computer. On the other hand the occasional constraints on memory are often severe enough to be major bottlenecks in human performance. The processes that make use of all the information stored in memory are recognition and memory search. Recognition is related to problem sclving to the extent that stimulus elements in the problem space suggest appropriate things to do. Memory search is involved in problem solving when more devious pathways must be taken in constructing a problem space, or in applyng problem-solving operators.

This chapter discusses how the information is acquired and processed, which is followed by the presentation of a cognitive model of memory. Finally memory classifications will be analysed and techniques for increasing the memory capacity will be discussed.

B. ACQUISITION OF INFORMATION

The human being depends on the environment where he lives and it is in this environment that he obtains the information needed for his survival. The sense organs are importants factors in this acquisition because they furnish a physiclogical representation of the outside world. An attention mechanism will select the conspicuous aspects of this representation for further processing by a central system. However, the nervous system introduces alterations in the physical image received, simplifying the information that must be transmited to high level analysing systems and later to the memory.

The central processing of this information can be executed in two different ways[Refs. 9,10]:

Bottom-up systems or data driven. The input information is treated in sucessive and increased levels of sophistication until the final recognition of the input.

Top-down systems or conceptual driven. This process starts with the highest-level of expectation of an object that is further refined by analysis of the context to yield expectation of particular lines in particular locations.This is a more powerful process than the bottom-up but it's strongly dependent on the ability to make syntactic choices of the objects to expect.

Top-down and bottom-up processing take place simultaneousely and come together in the job of the comprenhension of the outside world.

C. PROCESSING AND STORING INFORMATION

One of the aspects of the human thought process, related with computer programming, is the way the memory works and the information is processed and stored. A memory cognitive model commonly adopted[Ref. 6] is depicted in Figure 3.1.

In this model very-short-term-memory (VSTM) is composed of locations to hold data for a short time[Ref. 9]. This information can be retrieved by the short-term-memory (STM) by an attention mechanism. Here another process occurs (perception or recognition) related with the analysis of the individual characteristics of the stimulus and the context where these characteristics are inserted.

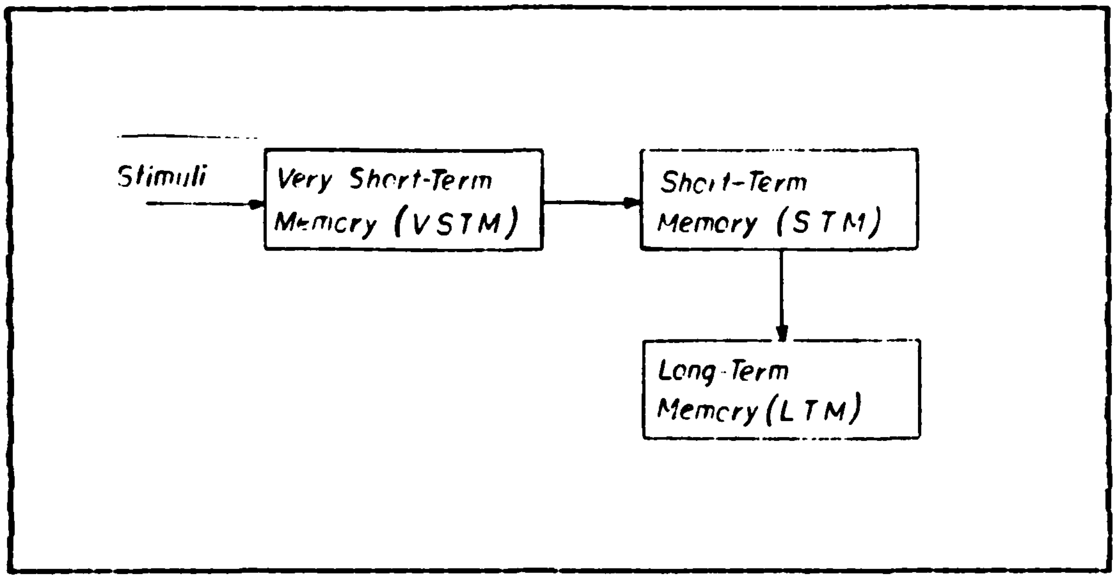


Figure 3.1 Memory Cognitive Model.

The STM has a temporary and limited capacity to store information. Its span imposes severe limitations on the amount of information that we are able to receive, process and remember. Miller[Ref. 11], in his paper "THE MAGICAL NUMBER SEVEN PLUS OR MINUS TWO" identifies 5-9 chunks of information as the capacity of short-term memory. This information is highly volatile and can be lost by the changing of attention. To avoid this problem it will be necessary to rehearse the information. The reherasal process consists of refreshing the contents of STM by continuous repetition to oneself.

Finally, in this process, the information needs to be stored in a permanent place called long-term-memory (LTM). The LTM is characterized by its unlimited capacity to store the programmer's permanent knowledge. The store process is relatively slow and requires a second rehearsal for fixing this information (learning).

D. MEMORY IN PROBLEM SOLVING MODEL

In problem solving processes it will be necessary to introduce modifications in our model[Ref. 12]. Following Feigenbaun new components will be incorporated as shown in Figure 3.2.

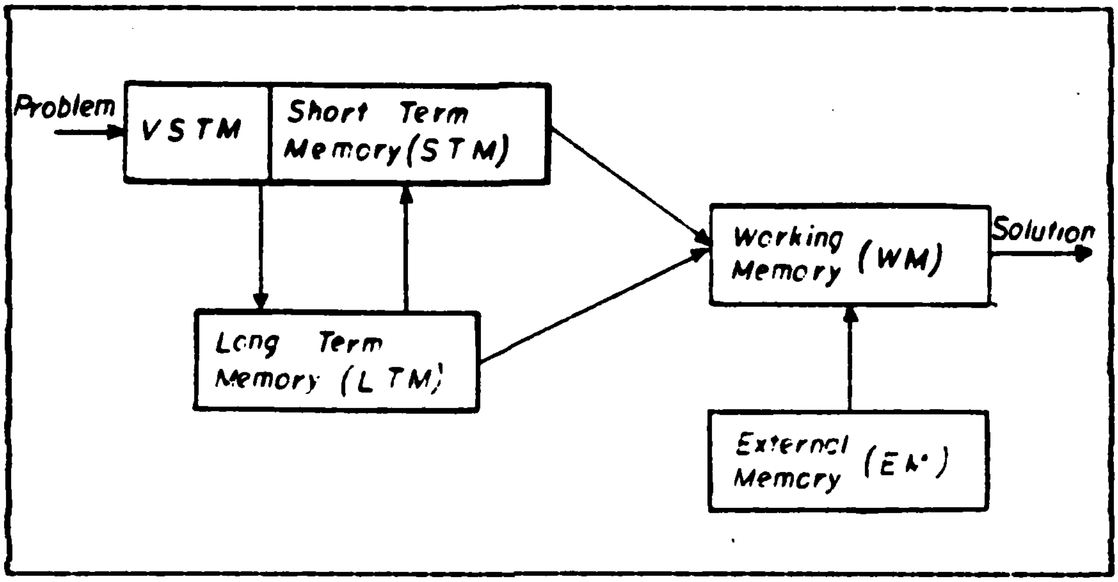


Figure 3.2 Components of Memory in Problem Solving.

These new components are the working memory and external memory. The working memory is characterized by having more permanent storage capacity than STM and less than LTM. The working memory plays the role of integrating all the information from the STM and LTM, of analyzing data, of building it into new structures and furnishing the results to be used to generate solutions.

The external memory collects all the information contained in external sources (modules,models,programs, documentation) and is helpful to develop possible solutions to the problem[Ref. 13]. It also compensates for the slow fixation times associated with the LTM, and frees the limited STM resources for use in problem solving (creativity, concentration etc.).

E. PROBLEM SOLVING TASKS

The process related with problem solving tasks involves the following steps[Refs. 6,10]:

- Program composition
- Comprehension and design a solution
- Coding
- Debugging and modification
- Learning

1. Program Composition

In this first step the problem is presented to the programmer. By a memory mechanism it passes from the short term memory to the working memory. Here the problem is analysed and defined in terms of the "given state" and "goal state". At the same time additional information is called from long term memory and external memory for further analysis.

2. Comprehension and Design of a Solution

This second step is one of the most important because it is the basis for debugging, modification and learning tasks. The programmer constructs a multilevel internal semantic structure (hierarchical) with the aid of his syntactic knowledge of the language. At the top of this hierarchical structure the programmer develops a comprehension of what the program does. At the lower levels the programmer may recognize the algorithms or common sequences of statements that can be used to solve the problem (solution). The important issue here is that the programmer develops an internal semantic structure for representing the syntax of the program, but he doesn't need to memorize or comprehend the programm line-by-line based on the syntax.

3. Coding

In this third step, the programmer will translate the program to internal semantic structure using an encoding process similar to chunking. The programmer will recognize the function of groups of statements instead of character-by-character, and chunk this group of statements into progressively larger chunks until all of the program is comprehended and the internal semantic structure is developed. Then the programmer could convert the program to any programming language and explain it to others easily.

4. Debugging and Modification

In debugging we are going to identify the errors that can occur in the composition task. These errors result from an incorrect transformation from the internal semantics to the program statements or from an incorrect transformation of the problem solution to the internal semantics. The first kind of error can be detected by analysing the output which, in case of error, will differ from the expected output. These errors can be originated by mistakes in the coding of a program or from incorrect knowledge of the functions of certain syntactic constructions in the programming language. The second kind of error is more difficult because their recovery implies a total reevaluation of the programming strategy. They are, for example, failure to deal with out-of-range data values, inability to deal with special cases such as the average of a single value, etc.

Modification develops by two steps. The first step consists of understanding the internal semantic structure of the program to modify. The second step consists of changing this semantic stucture in function of the modification needed with the consequent alteration of the programming statements. This is a complex task that requires knowledge in composition, comprehension and debugging.

5. Learning

This last task consists of the acquisition of new programming knowledge. The two classes of knowledge, semantic and syntactic, are acquired in two different ways. The semantic knowledge is acquired by meaningful learning through the development of internal semantics for a particular problem, and it is essential during the problem analysis. The syntactic knowledge acquired by rote learning is specific to the language used, and becomes important during the coding and implementation phase.

F. MEMORY TRACES CLASSIFICATION

The memory traces can be classified as non-associative and associative memories[Ref. 14].

1. Non-Associative Memories

This kind of memory consists of records encoded and stored in locations (cells,registers,etc.) in the order that they occur. Its purpose is to get the exact temporal sequences of the events. In computer terminology this representation is usually denoted "location adressable" because we can obtain directly the contents of a particular location to answer questions. In non-associative memories we can have one dimensional non-associative memory as for example the sucessive sections of magnetic recording or the columns of an IBM card, or two dimensional non-associative memories such as charts, tables or pictures. The human memory involves non-associative memory when it creates external memory (documentation, tables, modules etc.).

2. Associative Memories

Associative memories consist of records of events that are encoded and stored by networks of nodes. The big difference between this type of memory and non-associative memory is that when the same event occurs at a later time, precisely the same node or set of nodes are activated (direct access). This constitutes an important economy in the representation of events.

The human conceptual (semantic) memory involves association of particular concepts, events, facts and principles with each other, but to retrieve information, memory must be given specific cues.

3. Hybrid Memories

The computer memories are not as fully associative as the human memory. One can tell that it is hybrid because it is a combination of associative and non-associative memories. The information (documentation) is stored in a non-associative manner but each of these documents will be indexed by a large number of items and any of the various combinations of indexing terms will provide relatively direct access to the document through a sorting tree that works as an associative memory.

G. VERTICAL ASSOCIATION OR CHUNKING

Given the severe capacity limitations of short-term-memory, one method of reducing these limitations and so expanding our capacities is by "chunking"[Ref. 11].

As commonly used this term refers to regrouping or recoding the stimulus information presented.For example if the unbroken seven-item 4731052 was translated into 473 pause 1052 one would have one type of chunking (regrouping) cr if 11010000011 (binary) was translated into 6403 (octal) one would have another type of chunking (recoding). The importance and usefulness of chunking was first sugested by Miller and as experimental evidence he actually used a demonstration similar to the binary octal translation example given above. Here two main points about chunking in short-term-memory are shown. First, memory as measured by memory span is more a function of the number of chunks of information, than the number of bits of information. Second memory span, for binary digits, could be dramatically increased by a recoding technique. Miller also points out that memory span is primarly a matter of the number of chunks we can recall, regardless of the amount of information contained in each chunk.

B. EXTERNAL MEMORY

External memory, one of the components of human information processing, can be viewed in two different ways depending on the type of aid that it can furnish and its application in the programmer's work. The first one, external aids in domain reconstruction, will be analyzed in Appendix A and the second, external aids related with the operation of an interactive computer system, will be discussed in Appendix B.

IV. KNOWLEDGE ACQUISITION

A. INTRODUCTION

This chapter outlines the basic conceptual understanding of computer programming process and the knowledge-based approach used for its development. The ideas outlined here are embodied in a tool intended to implement a radically new software process. This new tool (reusability of programs) becomes each day a more important way to solve the actual problems of generation of new software.

B. SYNTACTIC/SEMANTIC KNOWLEDGE

The knowledge stored in LTM can be divided into two different parts [Ref. 6]: Syntactic and Semantic Knowledge Figure 4.1.

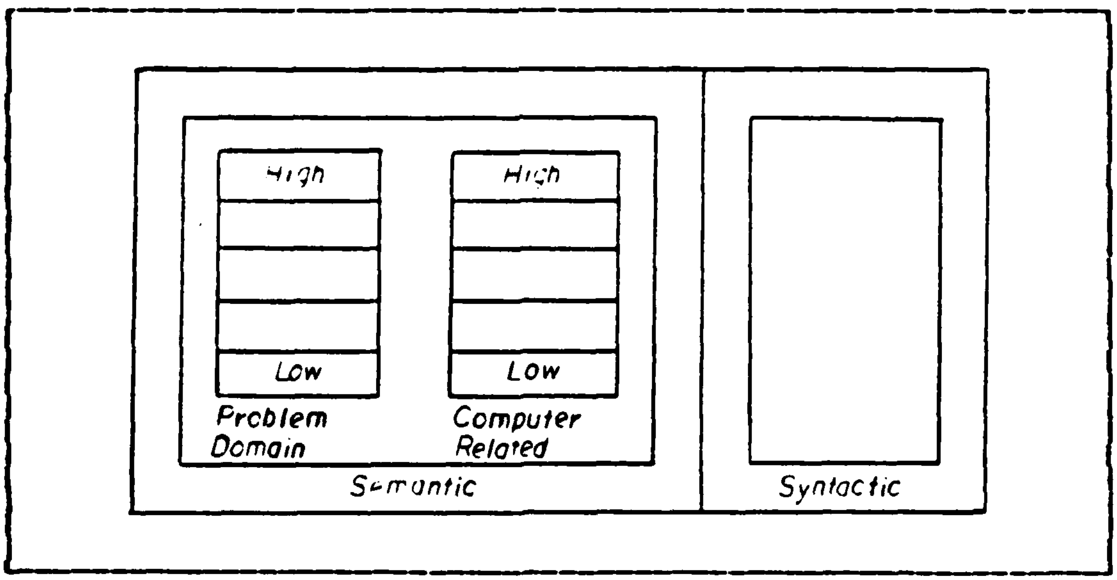


Figure 4.1 Knowledge in Long-Term-Memory.

1. Syntactic Knowledge

The syntactic knowledge is characterized by its precision and detail and involves the knowledge of the structure of the language, formats, iteration, conditionals, assignment statements, libraries of functions, etc.

2. Semantic Knowledge

Semantic knowledge is located in LTM and it has two components: computer related concepts and problem domain concepts. Semantic knowledge has a hierarchical structure going from low-level action to high-level goals.

3. Computer-Related Concepts

Computer-related concepts include objects and actions at high and low levels. For example, a central set of computer-related object concepts deals with storage. Users come to understand the high level concept that computers store information[Ref. 6]. The concept of store information can be refined into the object concepts of the directory and files of information. In turn the directory object is refined into a set of directory entities which each have a name, length, data of creation, owner,acess control etc. The file objects can be decomposed into program files, data files, index files, text files, image files, audio/speech files etc. Each file may have a lower level structure consisting of lines, fields, characteristics, pointers, binary numbers etc.

The computer-related actions with respect to stored information include saving and loading a file. The high-level concept of saving a file is refined into the middle level actions of storing a file on one of many disks,of applying access control rights (or simply write protections in most cases), of overwriting previous versions, of assigning a name to the file, etc. Then there are many low-level details about permissible file types or sizes, error condition such as shortage of storage space, or responses to hardware or software errors.

Users can learn computer-related concepts by seeing a demonstration of commands, hearing an explanation of features, or by trial and error. A common practice is to create a model of concepts, either abstract, concrete, or analogical, to convey the operation. For example, with the file saving concept, an instructor might draw a picture of a disk drive and a directory to show where the files go and how the directory references the file. Alternatively the instructor might make a library analogy and describe how the card catalog acts as a directory for books saved in the library.

Since semantic knowledge about computer-related concepts has a logical structure and since it can be anchored to familiar concepts, this knowledge is expected to be relatively stable in memory. If we remember the high level concepts about saving a file, we are able to conclude that the file must have a name, a size, and a storage location. The linkage to other concepts and the potential for a visual presentation support the memorization of this knowledge.

In conclusion, the user must acquire semantic knowledge about computer-related concepts. These concepts are hierarchically organized, can be acquired by meaningful learning or analogy, independent of the syntactic details, hopefully are transferable across different computer systems, and are relatively stable in memory.

4. Problem-Domain Concepts

The usual way for people to deal with large and complex problems is to decompose them into several small problems, in a hierarchical manner, until each subproblem is manageable. Thus, a book is decomposed into chapters, the chapters into sections, the sections into paragraphs, and the paragraphs into sentences.

Similarily, problem domain actions can be decomposed into smaller actions. As an example in writing a business letter with a computer the user has to integrate three forms of knowledge. The user must have the high-level concept of writing a letter (problem domain), recognize that the letter will be stored as a file (computer related domain) and know details of the save command (syntactic knowledge). The user must be fluent with the middle level concept of composing a sentence (problem domain), recognize the mechanism for begining, and ending a sentence (computer-related) and know the details of how sentences are demarcated in the screen (syntactic knowledge). Finally the user must know the proper low-level details of spelling each word (problem domain), comprehend the motion of the cursor on the screen (computer-related domain), and know which keys to press for each letter (syntactic knowledge).

Integrating the three forms of knowledge, the objects and actions, and the multiple levels of semantic knowledge is a substantial challenge which takes high motivation and concentration. Learning materials that facilites the acquisition of this knowledge are dificult to design, especially because of the diversity of background knowledge and motivation levels of typical learners. The syntactic/semantic model of user knowledge can provide a guide to educational designers, by highlighting the different kinds of knowledge that users must acquire.

C. KNOWLEDGE DOMAIN

A great number of tasks in computer programming and software reuse are closely related to the programmer knowledge that is critical for understanding, testing and debugging a program and in the development and maintenance of the software.

This knowledge can be seen as a succession of knowledge domains which bridge between the problem domain language and the final problem domain, execution Figure 4.2.

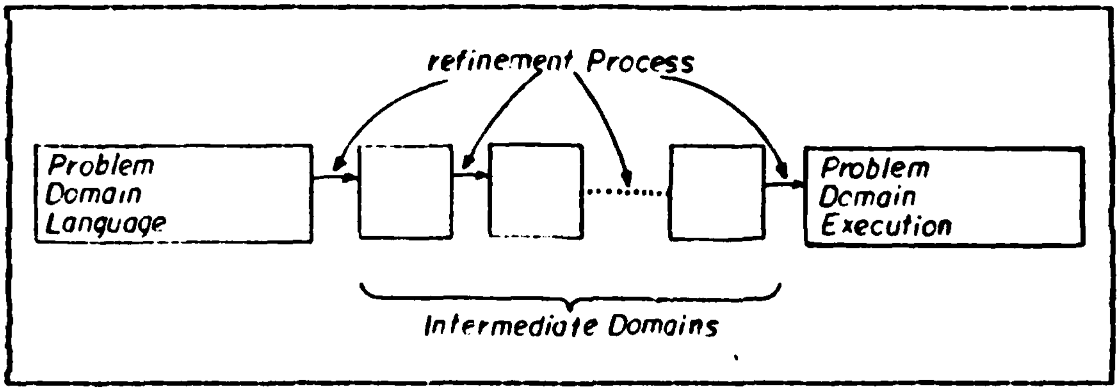


Figure 4.2 Knowledge Domains in Problem Solving.

Ruven Brooks[Ref. 13], presents a theory of how the understanding phase is acomplished and how it is based on the concept of knowledge domain. This concept is defined as a set of primitive objects, properties of the objects, and relations among objects and operators which manipulate these properties or relations. Following this theory the task of developing a program consists of constructing and reconstructing information about the modelling "knowledge domains" beginning with the program in execution.

This concept of domain provides a convenient encapsulation of one problem in the following way: the problem is presented in one domain language. When a refinement process is invoked the problem passes through one or more intermediate domains, ending in the execution of the program. Also it is important to present the concept of the refinement process. This concept consists of restating the problem specified in one domain into other domains by using or excluding assertions. The choice of the refinement process will have to obey and maintain the consistency of the developing problem but its level of abstraction must be reduced.

D. DOMAIN ACQUISITION

The acquisition of a knowledge domain can be viewed as acquiring two different types of information. First the programmer has to know the set of objects within each domain, their properties and relationships, the set of operations performed on these objects and the sequences in which they occur.

The second is related to the information about the relationships between objects and operators in one domain and those in a nearby domain.

To acquire this knowledge, the programmer has tc use different sources of information contained in the program (for example, variables, structure, procedures etc.) and external aids such as user's manuals, flowcharts, program design languages, that will be analyzed in Appendix A.

E. DOMAIN RECONSTRUCTION

Now synthestizing the several concepts presented before, we cansee the two different processes to understand a program known as the data driven and concept driven processes. The first one, which is more naive, uses a bottom-up hierarchy where the programmer tries to understand each line of code and assign them interpretations. Then he aggregates these interpretations to provide the understanding of larger segments of code. In the second process, based on a top-down hierarchy, sucessive refinements of hypotheses from other knowledge domains will be performed and their relationships to the execution of the program established.

These hypotheses appear from the person's knowledge, the task domain and the other domains that might relate to it. The refinement process is progressive and interactive and is based on the information extracted of the program text and other sources and can involve generation of subsidiary hypotheses. With this hypothesis and certain features of the program text, the programmer can reconstruct the knowledge domain for a particular job that is being performed.

Finally we can use the procedure to acquire information to reconstruct the knowledge domain in the following way:

When the programmer obtains any information about the program or its description a primary hypothesis is created. Then, by a process of verification the programmer generates sucessive subsidiary hypotheses in a top-down, depth-first manner (hypothesis hierarchy generation) that will be refined. The lowest point in this hierarchy may be refined enough to be verified against the program text or documentation.

F. DOMAIN KNOWLEDGE AND REUSABILITY

Developing domain knowledge theories is difficult, but theories can be designed in such a way as to be reusable[Ref. 15]. Reusable domain theories can be viewed as nodes in a network. The direct arcs indicate the directions of ontological shifts that explain concepts in one theory in terms of concepts in other theories. These logical links are developed as steps along abstraction dimensions of classification, aggregation-decomposition and generalization-specialization [Ref. 16].

The conceptual modelling activity produces a parallel development of a domain language network. Entities, relations, functions etc. in domain theories have corresponding constructs in the domain languages. Their implementation corresponds to the translation functions of the theory network and reflect the abstraction processes used. By defining a network at a high level with respect to domain languages, we are separating the domain modelling problem (using a syntactically decoupled language) and the model integration problem. The network (unlike most wide spectrum languages) is neutral with respect to modelling application knowledge and effectively implements extensible families of languages. The orthogonality of the domain languages enable the implementation of projection mechanisms allowing the system developer to view a system from different perspectives at any point in its evolution [Refs. 16,17].

V. REUSABILITY

A. INTRODUCTION

Software reusability can be defined as the extent to which software products can be used in other applications. Reusabilty is measured in terms of the effort required to move a software product or a part of a software product to another application.

Reusability is a very important concept in software engineering and invclves a large scope of actions directly related to the programmer, his behavior and the organization of his knowledge.

In this field we can consider two different ways to acomplish this task. For the first one the problem is presented as a set of needs which potentially can be solved by a software program. Then the programmer attempts to meet those needs by creating a semantic knowledge model cf the problem. Finally with a knowledge of software workproducts from previous development situations, he incorporates one or more of those workproducts in the creation of the new program. This is the common way to make software reusable.

In the second way the programmer acquires a large knowledge of the software programming process by studying pieces of software already tested, that are available from external aids (external memory). Then the programmer is able to construct a semantic model in his mind and easily to translate it to code. To accomplish this task he needs a syntactic knowledge which is specific to the language that he will use. This is the traditional process to produce software and we will refer to it as "software reconstruction". That is, the programmer using his knowledge base and external memories "reconstruct" the program from his mind.

Both ways involve the principles presented in the last chapters. We can see how the human process is developed and the fundamental role of the memory mechanism and attention in the process. The new theories of cognitive science bring important help to understanding how the comprehension task is executed and how the knowledge is stored in memory. The cognitive model presented by Shneidermann and Mayer completes this ideas and clarifies the process of the human thinking.

The reusable task development begins by the comprehension of the problem to be solved, using the problem solving model depicted in Chapter II. Then the programmer was to acquire the whole set of related information, which constitutes the set of several domain knowledge involved, and constructs his semantic knowledge. After this the programmer chooses the best approach to solve the problem.

The cognitive theory provides a more sophisticated model of how people reuse software products. The model shows that in some situations the programmer may use the results of previous projects to reconstruct a new product. Thus the previous software product has made a significant contribution to the programming process, but this is not called reuse because the previous product was not copied into the new product. This suggests a reason why reuse is not used more widely and suggests that reuse may not be ever used as extensively as some proponents avocate.

B. CHARACTERISTICS OF REUSABILITY

Reusabilty of software requires the software be understandable, flexible, modifiable, and accessible. Simplicity, systems clarity and self descriptiveness criteria will enhance the understandability. Generally, machine and software independence, application independence and modularity will improve the flexibility, modifibility and adaptability. Well structured documentation and machine independence were consolidated into and replaced by the term independence.

The reuse of program products has a number of obvious payoffs such as reduction of costs,increased reliability, increased performance and enhancement of software systems. If the effort required to reuse the software is much less than that required tc implement it initially and the effort is small in an absolute sense,then the software program is highly reusable. The degree of reusability is determined by the number, extent and complexity of the changes, and hence by the difficulty in the software implementation process.

C. PRINCIPLES OF REUSABILITY

It will be useful to present some concepts that are very important to consider in a reusable application. They are the basis of effective work in this field.

1. Reusable Architecture

This concept is related to the necessity to create a specific architecture for reusability. Kendall pcints out[Ref. 18] that an effective reuse requires an architectural starting point,rather than joining modules and trying to link them together.

The approach presented by Kendall has the following attributes:

All the data description should be external to the programs or modules intended for reuse;

All the literals and constants should be external to the programs or modules for reuse;

The input/output control should be external to the program or modules intended for reuse;

The programs or modules intended for reuse consist primarly of application logic.

Even though this architecture is not complete(it does not deal with graphics,voice,or nonstandard data), this model is an important approach in the domain of reusability.

2. Modularization

Some software is reusable because it has been built to be sufficiently general to be adaptable to a sizable family of applications. This idea can be implemented in the concept to use modules in software reuse.

We can point to some factors advantageous for using this approach:

The possibility of handling modules as data;

Modules which are good abstractions and have general interfaces with the rest of the software;

The use of specific modules as software interfaces to different parts of the environment of the software.

We can define a module as a program or a group of closely related progams. The structure of a module is based on the principle of information hiding. Following this principle, systems details that are likely to change independently should be the secrets of separate modules. The only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change. Every data structure is private to one module; it may be directly accessed by one or more programs within the module but not by other modules. Any other programs that require information stored in module's data structures must obtain it by calling the module program.

Finally some of the goals of this module structure are:

The decomposition into modules brings a reduction of software costs by allowing modules to be assigned and revised independently;

Each module's structure should be simple enough that it can be understood fully;

It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behavior of the other modules;

It should be possible to make a major software change as a set of independent changes to individual modules.

Based on the goals above, the software will be composed of many small modules and organized into a structural hierarchy. Each nonterminal node in the tree is composed of modules represented by its descendents. This is the fundamental concept where the DRACO [Ref. 16] paradigm lies, as we will see below.

D. FORMS OF REUSABILITY

It will be useful to present and examine some of the actual applications where reusabilty has been shown to be successful.

1. Common Processing Modules

These modules are standard "black box" modules that execute generic program functions. They are characterized by having high cohesion (perform one specific function) and loose coupling (meaning that they pass only the data required from the invcking program). They return only their input, resulting data and a validity code. These characteristics assure reusability in a maximum number of applications[Ref. 19].

2. Macro Expansions and/or Subroutines

This is the cldest reusable software technique. It has been used in assembly level languages as well as high level languages and is well suited for modelling procedural abstactions. They have been used extensively in constructing program libraries of mathematical functions.

3. Packages

Packages are usually collections of routines that together execute a number of possible related services. Their behavior and operation principles are similar to mathematical functions. Examples of this packages include accounting packages, statistical packages, payroll packages, linear programming packages etc. They are written for specific applications that are well understood.

Packages generally have to be treated as monolitic entities. They are difficult to modify or embed in other systems. Most packages are insufficiently paramaterized and therefore have limited use as generic entities. They have a low level of reusability because they are strongly dependent on specific operating systems.

4. Compilers

Another example where the reuse concept is applied is in compiler development. The specification language for compiler-writing is BNF which is used to describe the syntax of the language. Once the BNF formalism is assumed, a parser generator program can be built. This digests a BNF specification of a language and automatically generates parsing tables. These tables, coupled with a simple algorithm, allow for the syntactic analysis of sentences. The final tool is the compiler-compiler. This allows for the specification of the source language, the object language, translation of source language into object language and other optimizations. Once the user has provided complete details to the compiler-compiler, part of a compiler is produced.

As we can see the compiler-compiler presents a high level of reusability because if we furnished the set of specifications of one source language it automatically produces a compiler for this source language.

VI. THE DRACO PARADIGM

A. INTRODUCTION

This chapter will present and discuss a mechanism called DRACO which essentially consists of a model where the reuse concepts are applied in construction of software systems. The fundamental purpose purpose of DRACO has been to increase the productivity of similar software systems, and its approach is based on the construction of software from reusable software components in a reliable way. The programs produced from these models are very efficient with the major optimizations done in the intermediate modelling languages[Ref. 16].

Basically three activities executed by DRACO can be pointed out:

- DRACO accepts a definition of a problem domain as a high-level domain specific language. For acomplishing this task it will be necessary to describe the syntax and semantic of the domain language;
- After the domain language has been described, DRACO accepts a description of a software system to be constructed as a statement or program in the domain langage;
- Finally, once a complete domain language program has been given, DRACO can refine the statement into an executable program under human guidance.

For a better analysis of the DRACO model, four major themes dominate the way DRACO operates: the analysis of a complete problem area (domain analysis), the formulation of a model of the domain into a special purpose, high-level language(domain language), the use of software components to implement the domain language, and the use of the source to source program tranformations to specialize the components for their use in a specific system.

1. Domain Analysis

Domain analysis differs from systems analysis in that it is not concerned with the specific actions in a specific system. It is instead concerned with what the actions and objects cccur in all systems in an application area (problem domain). This may require the development of a general model of the objects in the domain, such as a model which can describe the layout of the documents used. Domain analysis describes a range of systems and is very expensive to create. It is analogous to designing standard parts and standard assemblies for constructing objects and operations in a domain. Domain analysis requires an expert with experience in the prcblem domain.

2. Domain Language

A DRACO domain captures an analysis of a problem domain. The object in the domain language represents the objects in the domain and the operations in the domain language represent the actions in the domain. It is commonly accepted that all languages used in computing capture the analysis of some problem domain. Many people bemoan the features of the Fortran language; but it is still a good language for making straight line output of calculations, the type of computing high-energy physics has done for many years. This is not to say that FORTRAN is a good analysis of the domain of high-energy physics calculations, but it has its place[Ref. 20]. Domains are tailored to fit into the right place as defined by the uses in which man is interested in using computers.

3. Software Components

As discussed in Chapter IV, software components are analogous to both parts and assemblies. A software component describes the semantics of an object or operation in a problem domain. There is a software component for each object and operation in every domain.

Once a software component has been used successfully in many systems, it is usually considered to be reliable. A software component's small size and knowledge about various implementations makes it flexible to use and produces a wide range of possible implementations of the final program. The top-down representation (refinement history) of a particular program is organized around the software components used to model the developing program. The use of components does not always result in a program with a block structure chart in the form of a tree. Usually, as with programs written by human programmers, the block structure chart of the resulting program is a graph as shown in figure 6.1.

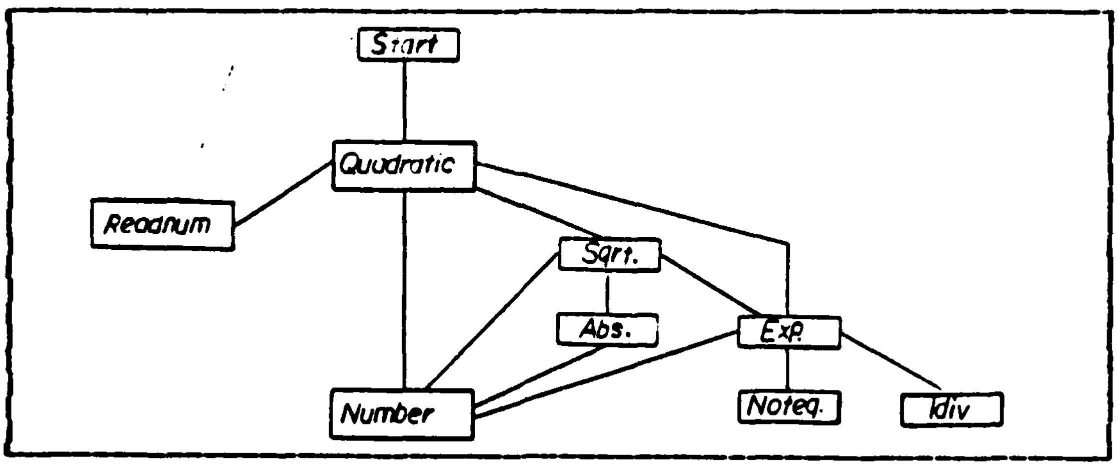


Figure 6.1 Block Structure Chart.

4. Source-to-Source Program Transformation

The source to source program transformation[Ref. 21] used by DRACO strip away the generality in the components. This makes general components practical. The tranformations also smooth together components, removing inefficiencies in the modelling domain. This makes small components practical. Since single-function, general components are esential to the parts-and-assemblies approach, the tranformations make component-built systems efficient and practical.

A tranformation differs from an implementation of a component (a refinement) in that transformations are valid for all implementations of the objects and operations they manipulate. Refinements can make implementation decisions which are limitations on the possible refinements for other components of the domain. In general transformations relate statements in one problem domain to statements in that same problem domain, while components relate statements in one problem domain to statements in other domains.

The DRACO mechanism, in this way can be considered as a general mechanism which can create (from human analysis) and manipulate (with the human guidance) a library of domains.

B. THE PARTS-AND-ASSEMBLIES CONCEPT

Among the several approaches to building things there exists one called "parts-and-assemblies" that has special importance for our study. The concept underlying this approach has been used extensively in engineering[Ref. 22] and it is one of the techniques which has enabled computer hardware engineers to increase the power and capacity of computers in a short time. The parts-and-assemblies approach relies on already built standard parts and standard assemblies of parts to be combined to form the object. This approach offers cheaper construction costs since the object is built from pre-built standard parts.

We can define an assembly as a structure of standard parts which cooperate to perform a single function. The use of standard parts and assemblies will supply some knowledge about the failure nodes and limits of the parts. This approach has as disadvantages that the design of useful standard parts and assemblies is a very expensive work and requires craftsman experience.

C. SOFTWARE CONSTRUCTION USING PARTS-AND-ASSEMBLIES

A software component is analogous to a part and can be viewed as either a part or an assembly depending on the level of abstraction of the view. The view of a particular component usually changes from a part to an assembly of subparts as the level of abstraction is decreased. This duality of a component is a very important concept and failure to recognize it caused some problems with earlier work on reusable software (representation of the software to be reused). In program libraries the programs to be reused are represented by an external reference name which can be resolved by an linkage editor. While the functicnl description of each program is usually given in a reference manual for the library, the documentation for a library program seldom gives the actual code or discusses the implementation decisions. The lack of information prohibits a potential use of a library program from viewing it as anything other than a part. If the user can treat a library program as an isolated part in his developing system then the program library will be useful. Mathematical function libraries fit well into this context.

Usually, however, a user wishes to change or extend the function and implementation of a program to be reused. These modifications require a view of the program as an assembly of subparts and a part of many assemblies. To decrease the level of abstraction of a library program in order to view it as an assembly of subparts requires information about the theory of operation of the program and implementation decisions made in constructing the program.

To increase the level of abstraction of a library program to view it as part of a collection of assemblies requires information about interconections between programs in the library and the implementation decisions defining common structures. None of this information is explicit in a simple program library; the burden is placed on the user of the library to extract this information.

Finally it seems that the key to reusable software is to reuse analysis and design, not code. In code the structure of parts which make up the code has been removed and it is not divisible back into parts without extra knowledge. Thus code can only be viewed as a part. The analysis and design representation of a program make the structure and the definition of parts used in the program explicit. Thus, analysis and design is capable of representing both the part view and assembly view while code only represent the part view. This is the fundamental principle of the DRACO approach[Ref. 16] for reusable software.

D. DRACO PARADIGM

The DRACO paradigm is used for the generation of software. In this approach one assumes that an organization wants to construct a number of similar software programs.

DRACO consists of an interactive system which permits a user to conduct the refinement of a problem stated in a high level problem domain specific language into an efficient, low level executable program. This is accomplished by making individual modelling and implementation choices and tactics, and by giving guidelines for semi-automatic refinement. Draco furnish mechanisms to enable the definition of problem domains as special purpose, high-level language with automatic translation into an executable format. The notation of these languages is the notation of the problem domain; it is not necessary for the user to learn a new language. When the user interacts with the system he uses the language of the domain.

E. AN EXAMPLE OF THE USE OF THE DRACO PARADIGM.

Supose an organization was interested in building many customized systems in a particular application area, say systems for aiding banks. They would go out to bank offices and study the activities of banks. A model of the general activity of being a bank would be formed and the objects and operations of the activities identified. At this point, the analyst of the domain of bank systems would decide which general activities of a bank are appropriate to be included in bank systems.

The decisions of which activities to include and which to exclude are crucial and will limit the range of systems which can later be built from the model. If the model is too general,it will be harder to specify a particular simple bank agency. If the model is too narrow, the model will not cover enough systems to make its construction worthwhile.

Once the analyst has decided on an appropriate model of bank activities, he specifies this model to the DRACO system in terms of a special-purpose language specific to the domain of banks and their notations and actions.

The idea here is not to force all the banks into the same mold by expecting them all to use the same system. If the model of the domain of banks is not general enough to cover the pecularities which separate one bank from another, then the model will fail.

The domain of bank systems is specified to DRACO by giving its external-form syntax, guidelines for printing things in a pleasing manner, simplifying relations between the objects and operations, and semantics in terms of domains already known by DRACO. Initially, DRACO contains domains which represent conventional, executable computer languages.

Once the bank domain has been specified, systems analysts trying to describe a system for a particular bank may use the model language as a guide. The use of domain-specific language as a guide by a system analyst is the reuse of analysis.

Once the specification of a particular bank system is cast in the high-level language specific to banks systems, DRACO will allow the user to make modeling, representation, and control-flow choices for the objects and operations specific to the bank system at hand. The selection between implementation possibilities for a domain-specific language is the reuse of the design.

Design choices refine the bank system into other modeling domains and the simplifying relations of these modeling domains may then be applied. At any one time in the refinement, the different parts of the developing program are usually modeled with many different modeling domains. The individual design choices have conditions on their usage and make assertions about the resulting program model if they are used. If the conditions and assertions ever come into conflict, then the refinement must be backed up to to a point of no conflict.

F. PRINCIPLES OF THE DRACO PARADIGM.

Before the program construction begins, the domain areas of interest are formalized by specification of each domain in the following way[Ref. 16]:

An (informal) set of concepts composed of objects, operators and relations;

A formal external notation for specifying an instance of the domain language;

A recognizer for the notation(parser);

A formal internal representation for the notation(an abstract graph constructed from the parser process);

A set of transformations which map internal representation in a domain to equivalent internal representations in that same domain (generaly used to effect optimizations).

A set of refinements which map individual concepts to one (or usually more) concepts in other domains.

The domains required to develop software for a given application area can be viewed as constructing a "domain structure graph" in which the nodes are domains and the set of refinements between them are represented as arcs. Such a network must provide for a refinement path to map high-level specifications into low-level implementations. Usually there are multiple paths through the domain network from an abstract domain node to an implementation domain node.

Software development starts with an abstract specification written using a combination of existing domain languages. The implementation process traverses a path through a space of possible implementations of progressively lower abstraction until a concrete implementation is reached Figure 6.2.

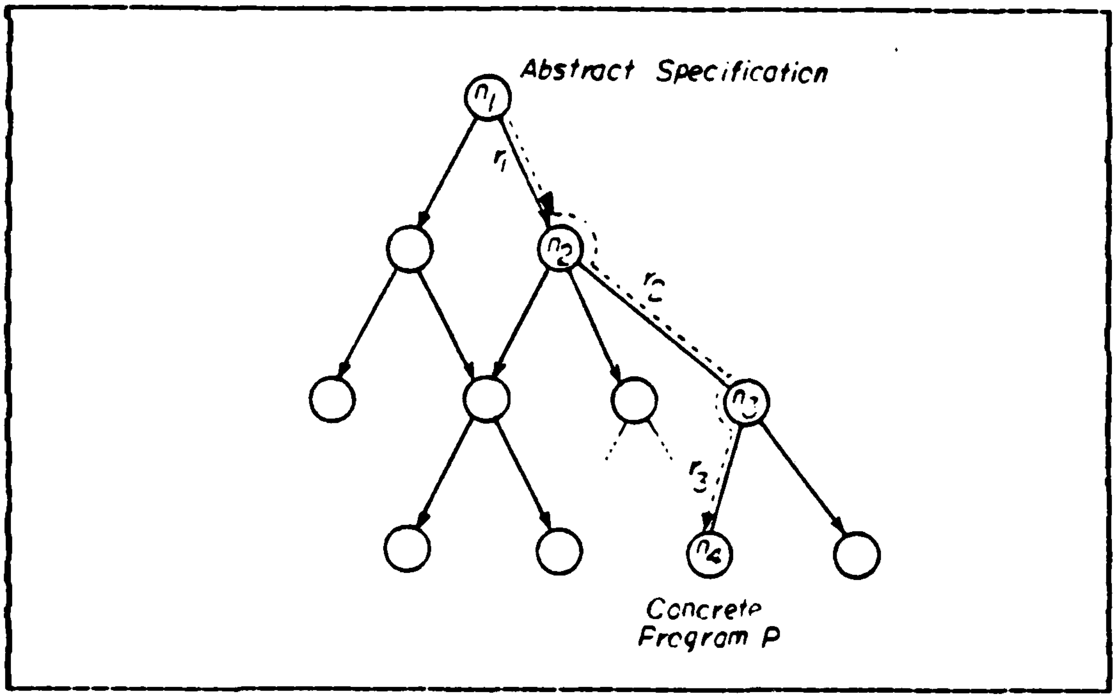


Figure 6.2 Construction of Program from Specification.

The space forms an enormous directed acyclic graph (DAG) called a "possible refinement DAG", with nodes in the graph representing specifications for the program written with notations from multiples domains. The single root of the DAG is represented by the initial specification. Leaves of the DAG are are concrete specifications. Arcs represent individual possible choices (refinements); the domains used by the specification at a node limits the type of arcs which exit that node to precisely those arcs emanating from the same domains found in the domain structure graph. Usually, an individual node is reached by many paths, representing different orders of choice of the same set of design decisions. A path from the root to a leaf represents a particular choice of a set of implementation design decisions and constitutes what is generally called the design. Navigation through the graph may be controlled by an implementation-style enforcing mechanism called tactics. Separate tactics can co-exist for different purposes: implementation for speed, for minimal space, for rapid prototyping,etc.

The refinement DAG is never constructed in its entirety. Only the path needed to reach a desired leaf from the root is explored. Once an implementation design path is chosen, it is not kept as such, but the design decisions that define the path are generally retained. A prototype tool to handle domain specifications and to construct an implementation path from abstract program specifications has been constructed by DRACO.

In Appendix C it will be shown how maintenance and recover of design in DRACO is acomplished.

VII. CONCLUSION

In this work the theories related to human thought processes, memory organization and the consequential implications on software construction are presented and discussed. Its importance in the new directions of programming development is obvious, since software reusability is one field where these concepts have primory influence.

The two approaches presented are conceptually different. The first one, more naive, represents the way reusability was understood in the past with its implementation based on the reuse of code. This form of software constuction represents the largest short-time payoff which explains why software producing organizations have been preoccupied with its utilization. However, it is very dificult to reuse code and it is not, in general, efficient because the specific analysis and design decisions are usually not obvious from reading the created code.

For the second, "software reconstruction", the software construction relies on the modern theories of domain analysis and design. The concept of knowledge domain is the keystone of this approach and its acquisition usually is difficult and expensive. The programmer has to spend a large amount of time in the acquisition of the knowledge involved because no one can be an expert in all the domains related with problem execution. Following this reasoning a programmer has to dedicate a long time to study the documentation contained in his external memory, to read all the literature involved and finally to construct the semantic model of the problem domain in his mind.

In conclusion, many of the future directions of software reusability will have to be based in this latter approach. Programmers should be instructed in this methodology because it is the way to create better software and at the same time to provide economic construction.

APPENDIX A

FLOWCHARTS AND PROGRAM DESIGN LANGUAGES

In computer programming it is very useful to have good techniques for representing a program because these techniques help the comprehension task and help in the debugging and modification tasks.

Among the actual possible representations of a program two of the most common and more controversial techniques will be presented: Flowcharts and Program Design Languages (PDL).

A. FLOWCHARTS

A flowchart consists of boxes containing instructions that are connected together by lines. Traditionally, flowcharts have been used as an informal notation for algorithms, but for more complicated algorithms flowcharts become intricate and dificult to draw and to follow.

Flowcharts were accepted for a long time for detailed program design documentation, but recently have been challenged with the argument that flowcharts may not aid program comprehension or error diagnosis and they are an unnecessary drain on project resources.

Knowledgeable programmers apparently prefer to work with the code itself rather than the lengthy detailed flowcharts. This is not surprising since a detailed flowchart is merely a syntactic recoding of a program and provides little additional aid. This coincides with the syntactic/semantic model of programmer behavior[Ref. 6] which sugests that a useful aid must facilitate encoding of the program syntax into higher level semantic units. An expert programmer deals more with problem domain related units than with program domain related syntactic tokens. High level comments using problem domain terminology have been shown to be more effective in aiding comprehension than numerous low level comments using program domain terminology.

These results and the syntactic/semantic model suggest that helpful documentation would provide a high level framework which reveals information that is dificult to obtain from the code itself. With a high level framework a programmer can anchor the knowledge acquired from reading each line or small unit of code.

B. PROGRAM DESIGN LANGUAGE

Flowcharts have long been accepted as the standard medium for detailed program design documentation. However several studies reported by Shneidermann et al.[Ref. 23] suggest that flowcharts may not aid comprehension of programs. Also, Ramsey and Atwood[Ref. 18] considers that a computer program expressed in a higher level language is more comprehensible than the corresponding flowchart. An artificially designed language, with a programming-language like syntax, might also be preferable to flowcharts for the expression of software design information. Such languages are commonly called program design languages(PDL's). Figure A.1 (From Kraly et al., 1975)[Ref. 24] shows an example of a PDL specification for a program which computes social security with holding (FICA) amounts from a payroll data base and prints a report of those values.

C. FLOWCHARTS VS. PROGRAM DESIGN LANGUAGES

The use of a PDL by a software designer for the development and description of a program design produced better results than the use of flowcharts[Ref. 25].

```
PRINT FICA REPORT HEADER
OBTAIN FICA PERCENT AND FICA LIMIT FROM CONSTRAINTS FILE
SET FICA TOTAL TO ZERO
DO FOR EACH RECORD IN SALARY FILE
    OBTAIN EMPLOYEE NUMBER AND TOTAL SALARY TO DATE
    IF TOTAL SALARY IS LESS THAN FICA LIMIT THEN
      SET FICA VALUE TO TOTAL SALARY TIMES FICA PERCENT
    ELSE
      SET FICA VALUE TO FICA LIMIT TIMES FICA PERCENT
    ENDIF
    PRINT EMPLOYEE NUMBER AND FICA TOTAL
    ADD FICA VALUE TO FICA TOTAL
ENDDO
PRINT FICA TOTAL
```

Figure A.1 An Example of a (PDL) Specification.

Specifically, the design appeared to be significantly better quality (involving more algorithmic or procedural detail), than those produced using flowcharts.

Flowchart designs exhibited considerably more abbreviation and other space-saving practices than did PDL design, with a possible adverse effect on their readability.

The information presented in these two media may be encoded in memory in different ways, at least with limited exposure time(Wright and Reid,1973)[Ref. 26], and the forms may differ in the processing effort required to encode them in memory even if they are encoded similarly.

PDLs and flowcharts may emphasize different properties of the underlying software design. At an obvious level, flowcharts appear to emphasize flow of control, while PDLs may have a greater emphasis on program structure.

Thus, in conclusion, an analytical comparasion of PDLs and flowcharts would appear, overall, to favor of PDLs for detailed design documentation. Only empirical evaluation, however, can provide really convincing evidence in favcr or one or another technique.

APPENDIX B

EXTERNAL AIDS IN OPERATION OF A COMPUTER SYSTEM

For the correct operation of an interactive computer system we have to have external aids like user's manuals and computer based manuals(online helps) which bring together all the information needed to operate a computer system.

A. TRADITIONAL USER'S MANUAL

The user's manual is a paper document that describes the features of the system. There are many variations in this theme such as an alphabetic listing, description of the commands, quick reference card with a concise representation of the syntax, novice user introduction tutorial and conversion manuals.

B. USER'S MANUAL DESIGN

The syntactic/semantic model offers insight into the learning process and therefore guidance for instructional material designers. If the reader knows the problem domain, such as letter writing but not the computer-related concepts in text editing and certainly not the syntactic details, then the instructional materials should start from the familiar concepts and tasks in letter writing, link them to the computer-related concepts, and then show the syntax needed to accomplish each task.

If the reader is knowlegeable about letter writing and computerized text editing, but must learn a new text editor, then all that is needed is a brief presentation of the relationship between the syntax and the computer-related semantics.

Finally if the reader knows letter writing, computerized text editing, and most of the syntax on this text editor, then all that is needed is a concise syntax reminder.

These three scenarios demonstrate the three most popular forms of printed materials: the introductory tutorial, the command reference and the quick review.

C. ORGANIZATION AND WRITING STYLE

To acomplish this task one must know about the technical contents, be sensitive to the background reading level and intellectual ability of the reader, and be skilled in writing lucid prose. Precise rules are hard to identify, but the author should attempt to present concepts in a logical sequence with increasing order of difficulty, to insure that each concept is used in subsequent sections, to avoid forward references, and to construct sections with approximately equal amount of new material. In addition to these structural requirements, the manual should have sufficient examples and complete sample sessions. Within a section that presents a concept, the author should begin with the motivation for the concept, describe the concept in problem domain semantic terms, then show the computer-related semantic concepts,and finally offer the syntax.

In summary we can present the following guidelines to help to write manuals:

Make the information ease to find.

Make information easy to understand:

-Keep it simple;

-Be concrete;

-Put it naturally.

Make the information task sufficient:

-Include all that's needed;

-Make sure it's correct;

-Exclude what's not needed.

Finally software and their manuals are rarely completed, rather they go into a continuous process of evolutionary refinement. Each version eliminates some errors, adds refinements, and extends the functionality. If the users can communicate with the manual writers, then there is a great chance of rapid improvement. Some manuals offers a tear-cut sheet for sending comments to the manuals writers. This can be effective, but other routes should also be explored: electronic mail, interviews with users, debriefing of consultants and instructors, written surveys, group discussicns, and further controlled experiments or field studies.

D. COMPUTER-BASED MATERIAL

In this type of aid we can consider the following types:

Online User Manual. An electronic version of the traditional user manual. The simple conversion to electronic form may make the text more readily available but more difficult tc read and absorb.

Online Help Facility. The most common form of online help is the hierarchical presentation of keywords in the command language, akin to the index of a traditional manual. The user selects or types in a keyword and is presented with one or more screens cf text about the commands.

Online tutorial. This potentially appealing and innovative approach makes use of the electronic medium to teach the novice user by showing a simulation of the working system by attractive animations and interactive sessions that engage the user.

Others forms of information acquisition includes classroom instruction, personal training and guidance, telephone consultation, videotapes, instructional films and audio tapes.

There is a great attraction in making technical manuals available on the computer. The positive reasons for doing so are:

Information is available whenever the computer is available. There is no need to go find the correct manual - a minor disruption if the proper manual is close by or a major disruption if the manual must be retrieved from another building or person.

User does not need to allocate work space to openning up manuals; Paper manuals can becomes clumsy and clutter up a workspace;

Information can be electronically updated rapidly and at low cost. Electronic dissemination of revisions ensure that out-of-date material cannot be inadvertently retrieved.

Specific information necessary for a task can be located rapidly if the online manual offers electronic indexing or text searching. Searching for one page in a million can usually be done more quickly on a computer than through printed material.

A computer screen can show graphics and animations that may be very important in explaining complex actions.

E. PAPER DOCUMENTS VS. ONLINE HELPS

The technology of printing text on paper has been evolving for at least 500 years. Much care has been taken with the paper surface, color, font design, character width etc. to produce the most appealing and readable format.

On the other hand the cathode ray tube (CRT) has emerged as an alternative medium for presenting text to meet user needs. Comparing these two media we can tell:

CRT display causes serious concerns about radiation and other health hazards such as visual fatigue. It makes the capacity to work with the CRT below the capacity to work with printed material.

It is easier to detect errors in printed text than the same text displayed in a screen.

Screens display substantially less information than a sheet of paper and the rate of paging through screens is slow compared to the rate of paging through the manual.

The reading rate is significantly faster on hardcopy (printed text) - 200 words/minute - than on the screen - 155 words/minute. Accuracy is slightly but reliably higher on hardcopy. The subjective ratings of screens are similar in both formats.

Still the online environment opens the door to a variety of helpful facilities which might not be practical in printed forms.

Some of these aids are:

Successively more detailed explanation of a displayed error message.

Successively more detailed explanations of a displayed question or prompt.

Explanation or definition of a specified term.

A description of the format of a specified command.

A display of a specified section of documentation.

Instruction on the use of the system.

News of interest to users of the system.

A list of available user aids.

APPENDIX C

MAINTENANCE AND DESIGN RECOVER IN DRACO

A. MAINTENANCE

We assume that a program has been derived from a specification using the DRACO paradigm and that the specification, the refinement DAG, and the implemented programs are all available to a would-be maintainer. We will discuss the maintenance problem in the absence of the specification and the refinement DAG in next section. Should a program need change, there are two methods for accomplishing it. One possibility is to choose an entirely new path through the refinement DAG from the initial specification to a different implementation. This method is generally not preferred, as many of the design decisions made for the current implementation can be reused in the desired implementation.

The other alternative is to start with the concrete implementation chosen, reverse some of the design decisions, moving up the refinement DAG towards the root, until a node is reached which is the last common abstraction (LCA) of the current implementation and the desired implementation. The least common abstraction is the top node of an embedded sub-DAG, and can be reached by any of several paths (as the design decisions need not be reversed in the order originally made). A new path must then be chosen from the LCA to the desired implementation Figure C.1.
This method preserves all of the implementation design decisions made above the LCA and thus minimizes work required to accomplish change.

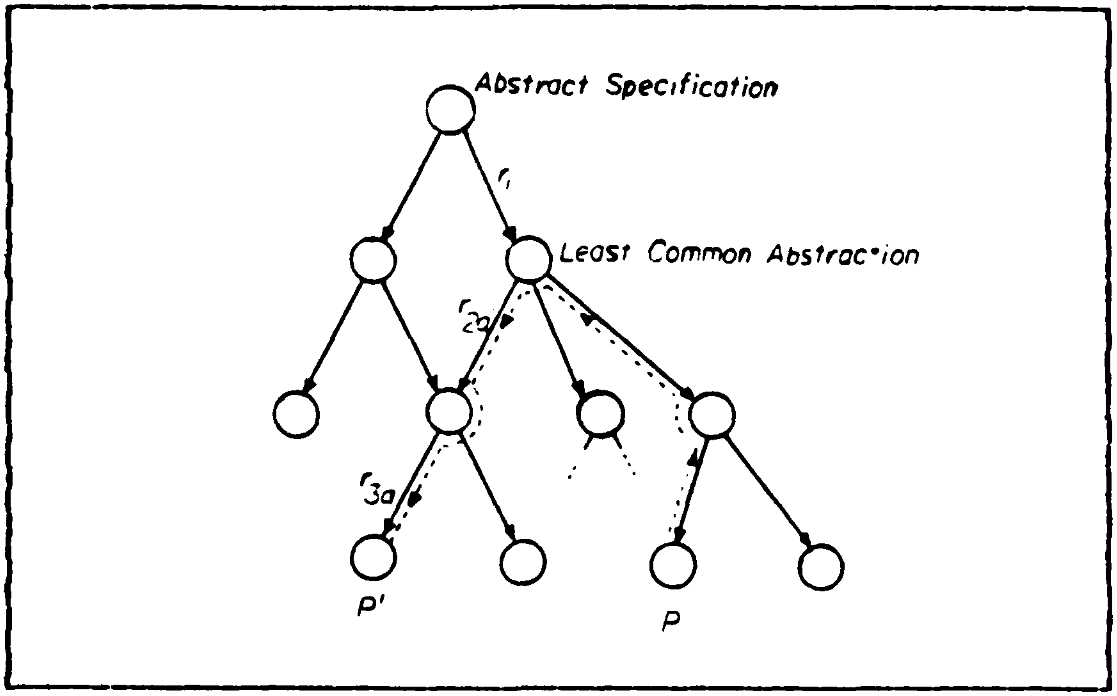


Figure C.1 Maintenance. General Choice r1 is Preserved.

Performance enhancement is generally acomplished by changing the underlying representations used by a program and using more efficient procedures made possible with the changed representation. We assume that the revised representations and corresponding procedures are already contained as refinements in the domains used to generate the current program (if this is not the case, then the domains must be augumented accordingly). Some set of nodes in the refinement DAG are LCAs that allow re-implementation of the currently low-performance abstractions. Design decisions are reversed to travel from the current implementation back to one of those LCAs. New decisions are applied to arrive at a different implementation. The change in refinement direction is accomplished by a change in tactics.

Changes in the environment can be handled in a similar fashion. The domains are first augmented with the refinement

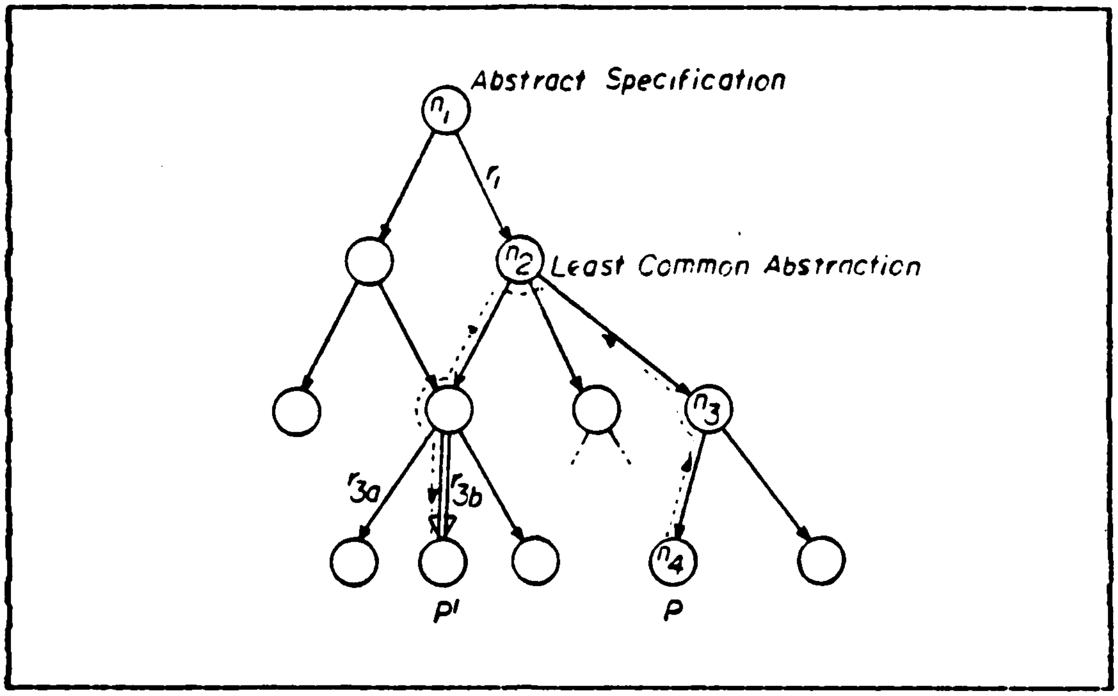


Figure C.2 Changing the Environment,r3b New Refinement.

specifying how the abstractions used in those domains can be implemented by the new environment; this effectively produces an implementation DAG Figure C.2. A suitable LCA is found and refined using the revised refinements. Different functionality is acccmplished by changing the specification. It is then straight forward, but possibly inefficient, to re-refine the specification to create a new refinement DAG different than the original.

A perhaps more efficient method for producing the revised program requires several steps Figure C.3:

> Determine a substitution S that converts the original specification to the revised specification (this can be constructed automatically as the origil specification is revised);

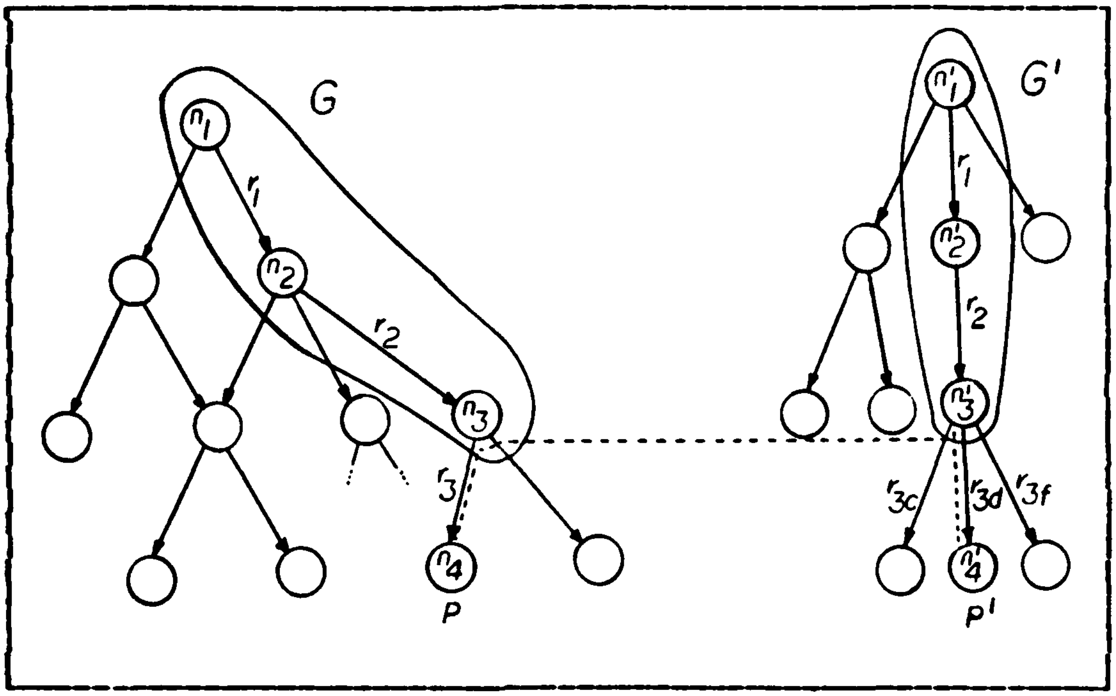


Figure C.3 Changing Specification. G"is Isomorphic to G.

Determine the largest subgraph G" of the new refinement DAG, starting in the top node, that is isomorphic with a subgraph G of the old refinement DAG under the substitution S. Each node n in G has a corresponding node n' in G", obtainable by applying the substitution S to n. Note that G' must include at least the root node (i.e., the revised specification).

Find an LCA of P in G. The corresponding node in G' can be refined to a concrete implementation P' which realizes the revised specification).

To determine the isomorphism, and therefore the candidate LCAs, the refinement DAGs need not be constructed in their entirety. The work accomplished in the original refinement history up to the chosen LCA in G can be reused at great saving. Refinements from the LCA in G" to the concrete implementation P' must be applied. This constitutes the bulk of the work. Design decisions used in the path from the LCA in G to P can perhaps be reapplied, reusing analysis done for the original program.

If the specification is modular, then there will be a refinement DAG for each part of the specification. The implementation will consists of a set of leaves, one taken from each DAG. A change to the specification will then affect only some of the specification modules, and so affect only some of the refinement DAGs. Leaf nodes from DAGs which do not change may be used unchanged in the new implementation. The procedure outlined above can be used to generate new leaves for the changed DAGs. Modularity is then seen simply as a method for making trivial the determination of the isomorphism on portions (the unchanged DAGs) of the what would otherwise be a single, large refinement DAG.

B. THE PROCESS OF DESIGN RECOVERY

In Figure C.4 we present a view of the conventional approach to maintenance. Arcs are represented by broken lines to indicate that the refinement history, and thus the original abstract specification, are not available. What is to guide the maintainer when going from program P to P'?

The DRACO paradigm offers a model of maintenance activities provided that the program specification and design are available. If we do not have these, we can recover them from the code, and then use the DRACO paradigm as the guide. The design recovery paradigm we propose provides a systematic way of carrying out the process that we think maintenance programmers apply informally: before performing changes in a program to adapt them to new requirements, a higher-level plausible "ancestor" specification equivalent to the original program is informally developed.

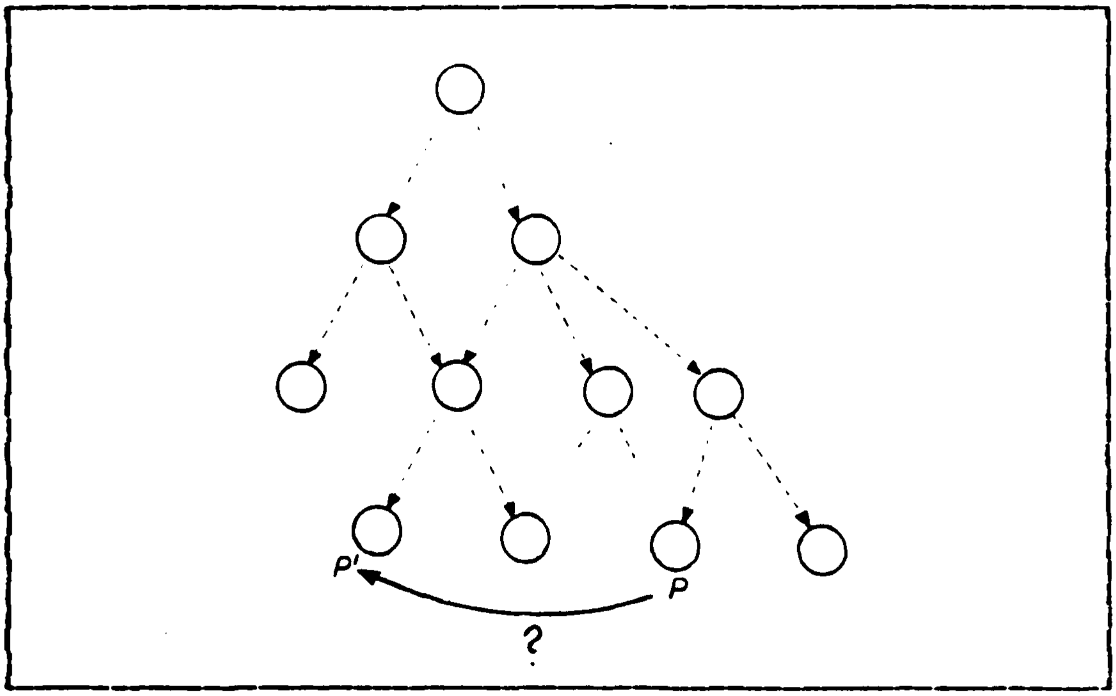


Figure C.4 Conventional Maintenance.

Such an ancestral specification can be developed by repeatedly performing a "design recovery step". Each step consists of inspecting the specification recovered from the previous step, proposing a set of possible abstractions of the portion of interest, choosing the "most suitable" abstraction, and constructing a specification containing the new abstraction. Each abstraction proposed implicity selects some domains and refinements which must produce the existing code when applied to the ancestor containing the proposed abstraction. Design recovery steps are repeated until a useful LCA is reached.

The design recovery process is ilustrated in Figure C.5. Starting with program P its plausible immediate ancestors (broken-circles) are postulated. Selection of an appropriate ancestor (solid circle) is based upon conjecture that the node is on the path from P to a suitable LCA.

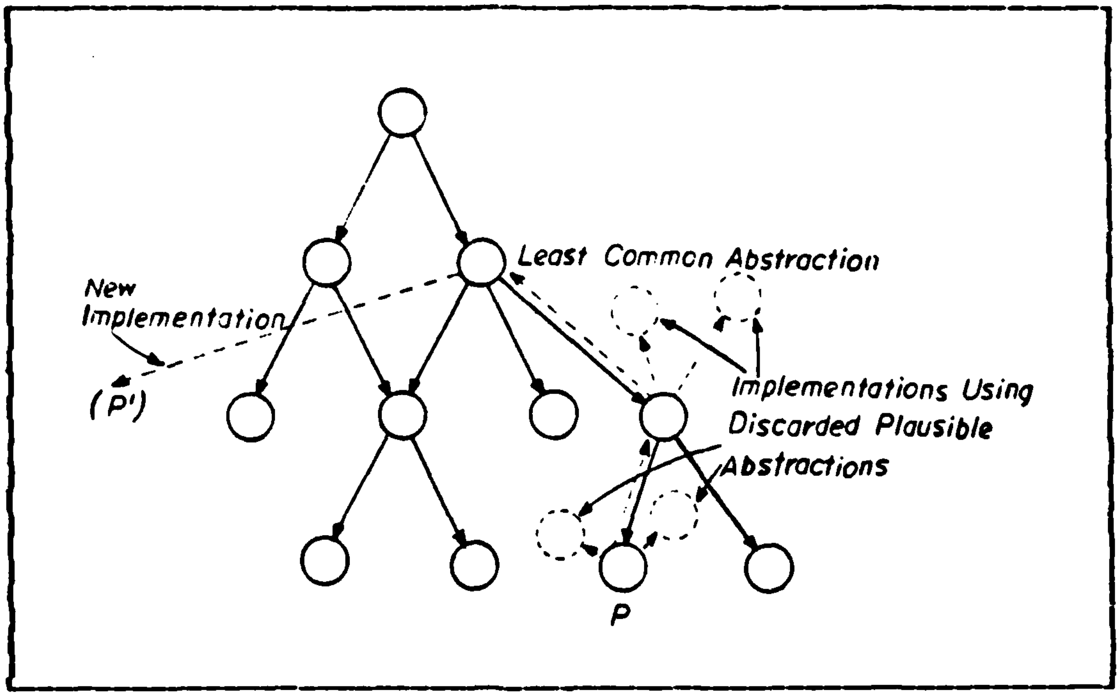


Figure C.5 The Process of Design Recovery.

Good choices of abstraction will use domains and refinements recovered in earlier steps, or will augment them minimally. The iterative process induces learning in the maintainer which can be captured in the resulting domains. The choice of the appropriate ancestor is the result of a generalization process based on the specification under consideration. The implementation provides a very limited sample on which to base a generalization step. In other words, refinements are possible only using additional knowledge: we must rely on the maintainer's knowledge of the application domain, intelligence, experience and educated guesses, on common knowledge and on any additional information available on the current implementation (e.g., inputs from original designer, existing documentation, environment specifications).

Since quite often the maintainers are not the original author, and are usually distant in time from the original implementation, maintainers are likely only to regenerate approximations of the original domains that where used. This mismatch between the maintenance DAG obtained by design recovery and an "ideal" Figure C.6 reveals the crux of the maintenance problem.

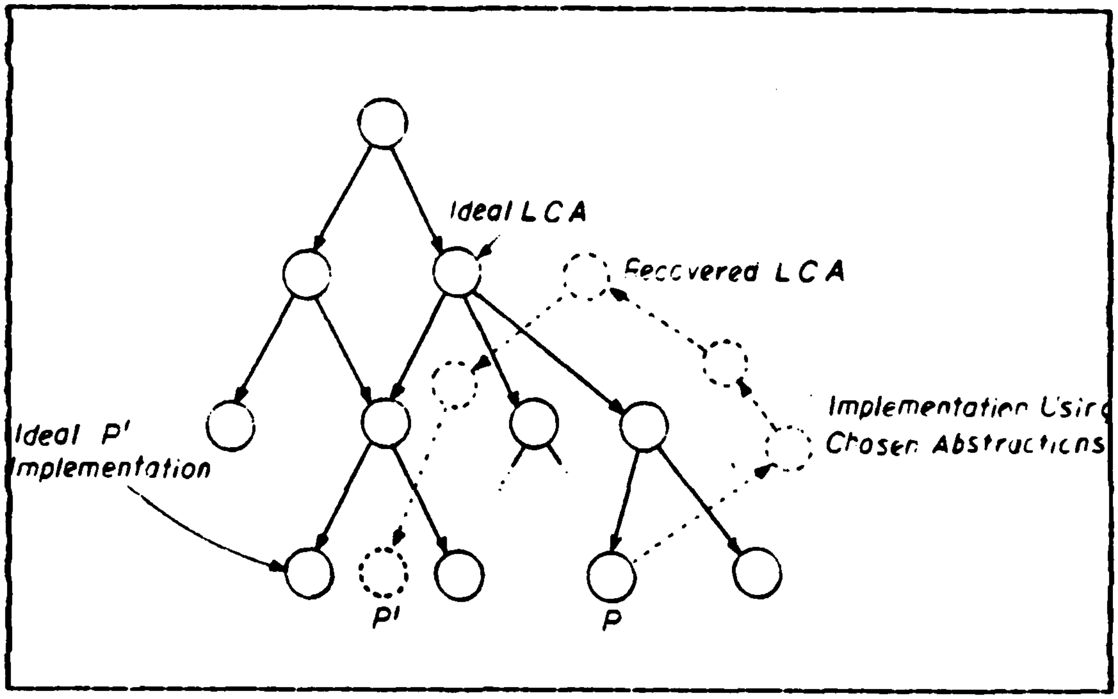


Figure C.6 Recovered Design vs "Ideal Design".

Avoiding approximations is very hard, and the approximation errors are typically amplified by repeated maintenance steps. The magnitude of the errors is increased when the recovery process is done informally. The errors, generated by the limited sample used for the abstraction step, can be substantially reduced by performing domain analysis.

Through domain analysis a more adequate, complete and reusable set of abstractions of a knowledge domain can be produced thus enhancing the power of the design recovery paradigm. This is the reason why domain analysis is a fundamental component of the DRACO technology.

LIST OF REFERENCES

1. Lehman,M.M.,Programs, Life Cycles, and Laws of Software Evolution Proceedings of the IEEE 68(9):1060 - 1076, September, 1980.

2. Moressey, J.H.,and Wu.L.S.,Software Engineering... An Economic Perspective ,In Procedings of the Fourth Conference on Software Engineering, pages 412-422, IEEE Press, 1979.

3. Boehm, B.,"Software and Its Impact: A Quantitative Assessment", Datamation 19(5) pages 48-59, Reprinted in Software Design Techniques by Freeman and Wasserman, May 1973.

4. Lientz, B.P., and Swanson, E. B.,Software Maintenance a User/Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organization, Reading, MA: Addison and Wesley, 1980,

5. Brooks, F.P.Jr.,"The Mythical Man-Mouth",Datamation 20(12): pages 45-52. Reprinted in Software Design Techniques by Freeman and Wasserman, December 1974.

6. Shneidermann, B., and Mayer,R.,"Syntactic/Semantic Interactions in Programming Behavior - A Model and Experimental Results", International Journal of Computer and Information Sciences, Vol 8, No 3, 1979.

7. Army Research Institute,Technical Report 392, An Exploratory Study of the Cognitive Structure Underlying the Comprehension of Software Design Problems, by M. E. Atwood, A. A. Turner, H. R. Ramsey, and J. N. Hooper, Alexandria VA, 1979.

8. Dijkstra, E. W.,Notes on Structured Programming, EWD 249, Technical University, Eindhoven, Netherlands,1969.

9. Norman, D.A.,Memory and Attention, John Wiley and Sons Inc., New York, 1976.

10. Tracz, W. J.,"Computer Programming and the Human Thought Process", Software-Practice and Experience, Vol. 9, pages 127-137, 1979.

11. Miller, G.A.,"The Magical Number Seven Plus or Minus Two : Some Limits of Our Capacity For Processing Information", Psychological Revue, pages 81-97, 1956.

12. Feigenbaun, E. A., Information Processing and Memory in Models of Memory, D.A. Norman Editor, Academic press, pages 451-469, New York 1970.

13. Brooks R., A Theoretical Analysis of the Role of Documentation in the Comprehension of Computers Programs, In Procedings of Human Factors in Computer Systems, New York : ACM, 1982 pages 125-129.

14. Wickelgren, W. A., Learning and Memory, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

15. Freeman, P., Reusable Software Engineering: Concepts and Research Directions, Procedures ITT Workshop on Reusability in Programming, pages 2-16, Stanford, C.T., 1983.

16. Neighbors, J., "The Draco Approach to Constructing Software From Reusable Components", IEEE Transactions Software Engineering, Vol SE - 10, pages 564-573, September, 1984.

17. Arango, G., A Knowledge Based Perspective on Software Construction, Submitted to IEEE Transaction on Software Engineering, 1984.

18. Kendall, R. C., "An Architecture for Reusability in Programming", ITT Programming, pages 1-3, Stratford, CT, 1983.

19. Cavaliere M. J., and Archambeaut, P. J., Reusable Code at the Hartford Insurance Group, Proceedings ITT, Workshop on Reusability in Programming, pages 273-278, September 1983.

20. Wegner, P., Directions in Software Techonology, MIT Press, 1979.

21. Kibler, D. F., Efficiency and Correctness of Transformation Systems, PhD Thesis, University of California at Irvine, 1978.

22. Edwards, N. P., and Teller, H., A Look at Characterization the Design of Information Systems, In Procedings of the A.C.M. National Conference pages 612-621, A.C.M., November 1974.

23. Shneidermann, B., Mayer, R., Mekay, D., and Heler, P., Experimental Investigations of the Utility of Detailed Flowchart in Programming, pages 373-381, Communications A.C.M. 20, 1977.

24. Rome Air Develorment Center, (NTIS No AD16415), Final Report (Report No. RADC-TR-74-300-Vol 8), Structured Programming series (Vol 8) : Program Design Study, by T. H. Kraly, J. J. Naughton, R. L. Smith, and N. Tinamoff, Griffiss AFB, New York, May 1975.

25. Research Institute for the Behavioral and Social Sciences, (ARI Technical Report TR-78-A22), Comparative Study of Flowcharts and Programming Design Languages for the Detailed Procedural Specification of Computer Programs, by H. R. Ramsey, M. E. Atwood, and J. R. Van Doren, Science Application, Inc., Colorado, 1978.

26. Wright, P., and Reid, F., "Written Information, Some Alternatives to Porpose for Expressing the Outcomes of Complex Contincences", Journal of Applied Psychology, pages 160-166, 1973.

BIBLIOGRAPHY

Bower, Gordon, Human Memory, Basic Processes, Academic Press, 1977.

Gregg, Iee W., Knowledge and Cognition, John Wiley & Sons, 1974.

Haugeland, John, Mind Design, A Bradford Book, MIT Press, 1982.

Jensen, Rendall W., and Tomes Charles C., Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey, 1982

Murdock Jr, Bennet B., Human Memory, Theory and Data, John Wiley & Sons 1974.

Neighbors, James M., Software Construction Using Components, University Microfilmes International, Ann Arbor, Michigan, 1980.

Pressor, Edward et. al., Software Interoperability and Reusability, RADC-TR-83-174 Vol II, Rome an Development Center, 1983.

Shneiderman, Ben, Software Psychology Human Factors in Computer and Information System, Little Brown Computer System Series, 1980

Wolberg, John R., Conversion of Computer Software, Prentice-Hall, Englewood Cliff, New Jersey, 1983.

Wickens, Christopher D., Engineering Psychology and Human Performance, Charles E. Mervill Publishing Company, 1984.

Young, John Z., The Memory System of the Brain, University of California Press, 1977.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Professor Gordon H. Bradley, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	4
4.	Professor Bruce MacLennan Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
5	Chairman, Code 052 Departament of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
6	Curriculum Officer, Code 037 Computer Techonclogy Program Naval Postgraduate School Monterey, Clifornia 93943-5100	2
7	Universidade Nova de Lisboa Praca Principe Real, 26 1200 Lisboa, Portugal	1
8	Direccao do Servico de Instrucao e Treino Edificio de Marinha Rua do Arsenal, 1188 Lisboa, Portugal	1
9	Comandante J. A. Cervaens Rodrigues Estado Maior da Armada - C.I.O.A. Rua do Arsenal 1188, Lisboa, Portugal	1
10.	Comandante Eduardo Manuel Pires Coelho Rua Nunes Claro 15, 1000 Lisboa, Portugal	3

END
DTIC
5–86