REQUIREMENTS ANALYSIS OF LARGE SOFTWARE SYSTEMS

Meir D. Burstin

July 1984
CSD-840048

REQUIREMENTS ANALYSIS OF LARGE SOFTWARE SYSTEMS

by

Meir D. Burstin

REQUIREMENTS ANALYSIS OF LARGE SOFTWARE SYSTEMS

Thesis submitted for the degree "Doctor of Philosophy"

by

Meir D. Burstin

Submitted to the Senate of Tel - Aviv University
July, 1984

This work was carried out under the supervision of
Dr. Moshe Ben-Bassat

This work is dedicated to my parents, Gina and Jacob Burstin.

ACKNOWLEDGEMENTS

I wish to express my gratitude to Drs. Dan Berry and Moshe Ben-Bassat for their guidance and their help.

A part of this research was carried out while I was a visiting scholar at the Computer Science Department at the University of California, Los Angeles. I wish to thank this department for hosting me, for enabling me to use their facilities, and for providing a warm and stimulating research environment.

I am grateful to Tadiran Inc. for letting me spend two years out of work doing this research.

I first encountered many of the problems and of the ideas that are the subject of this research during my years at Advanced Technology Ltd. I wish to thank all my friends and colleagues there.

Finally, I wish to thank my family, my wife and my children, for their patience, understanding, and constant encouragement. Without their help this work would not have been possible.

ABSTRACT

This thesis suggests a method to produce requirements for large software systems and provides a prototype of an expert system that assists in this method.

The method consists of seven stages: study, user identification, interviewing, raw requirement processing, verification, functional specification, and operational system specification. The basic unit of information in the suggested method is a requirement. It undergoes several modes of processing along the various stages of the method. Each requirement is generated, noted, classified, decomposed, composed with other requirements, checked, changed, and sometimes removed.

After an initial study of the needs of the specified system, a user identification process is initiated. An abstract user is defined as any entity that affects the system requirements. Abstract users are identified by a systematic hierarchical top-down decomposition. Next, the elementary abstract users are interviewed to form their requirements. A modified problem-solving technique is proposed for this stage.

In the next stage, implicit requirements are gathered and all the requirements undergo initial processing and classification. Then verification of the requirements occurs. This is done by abstraction of requirements of lower level users to higher levels and checking the result at each level. The next two stages convert the requirements into functional specifications and then into operational system specifications.

The requirements production method can be viewed as a process of transformations on system representations. First the system is represented by the net sum of the requirements of its users. Then during verification, it is represented by the net sum of the requirements of abstract users at various levels of abstraction. At the raw requirements processing stage and functional specification stage, the system is represented as being composed of main subjects and of functions. A different presentation is produced in the last stage. There, the system is represented in terms of basic operational procedures that are common to the specified system and its environment.

Each representation has different advantages. The user-oriented representation is best understood by the users and is easy to maintain for a system that is rapidly changing. The system implementors usually find the functional representation more natural and easier to work with. The customer, some of the users, and the operators all benefit from the operational representation.

The entire process can be computer assisted, and this automation is fully delineated in this thesis. A prototype was built that deals mainly with user decomposition, requirements composition, and verification. An expert system was developed to assist the specifiers in the composition of requirements that are stated in natural language.

Table of Contents

List of Figures

page

CHAPTER 1
Introduction

1.1 Requirement Specification

The requirements gathering phase is the first phase of a software project. Requirements are used as a contract, and as a communication medium among the system users and the system implementors, as criteria against which to check and verify the system after it is developed, and as a tool to launch changes and enhancements to the system. Their importance to proper development system is recognized today as a result of the many failures in projects that lack them, and awareness of this importance is increasing. Nowadays, it is obvious that good requirements are a necessary condition for having a good final product and for being able to manage the software project efficiently.

The study of requirements specification of a system has recently drawn a lot of attention in the academic community and in industry. It has become a research area of software engineering in its own right. Yet, not enough is known about the process to arrive at the requirements — about the best way to state them and to verify them.

The criteria to be used in judging software specifications are derived from their use [Bal81]. They should be understandable by all the parties that are involved in the project, they must be testable before the implementation is initiated and before the final product is used, and they must be easily modifiable.

A large software system is defined as a software system that has a large and diversified community of users and entails a variety of human, organizational and automated activities. A large software system tends to be physically very large in terms of lines of code, pages of documentation, etc. The development time and the life-cycle time of these systems are long, large amounts of money and other resources are needed during development and maintenance, and many users and parts of the organization affected by them.

There is one basic difference between a large software system as defined above and an ordinary software system. The requirements for an ordinary system are usually known in advanced or understood intuitively, and the whole system can be perceived by one person. However in a large software system, not all the requirements are known in advance; rather they have to be generated. Furthermore, knowing all the requirements does not necessarily mean understanding the system. The large size of the requirements makes delineation of the system difficult.

Formulating requirements for a large software system has been found to be very difficult for a variety of reasons, most of which may be attributed to two major factors. First, too many users with too many interactive requirements imply the difficulties inherent in the management of any large scale project. Second, a broad communication gap between the users and the developers, combined with non-coinciding objectives lead to the temptation to push under the carpet unclear or unresolved issues, knowing full well that they will pop up in the future. To this, add the perennial shortage of software professionals who are equipped with the unique psychological skills and experience required to conduct the interviews with the users, sort the structures, and formulate the output into a set of requirements.

A rigorous engineering method is required that enables the system users to formulate their requirements and to blend them into an entity that defines and describes the desired system. This method should provide means to check and verify the requirements against the needs of the users. The sheer size of a large software system leads to the need for a certain degree of automation in that process.

1.2 Overview

Many published articles discuss various ways to represent system or software specifications. These articles have defined specification languages, some of them are intended for requirements only and some are design languages that are hoped to be useful for requirements as well. It is quite difficult to draw a line between requirements and high level design, and therefore many articles tend to deal with both.

Several software tools have been designed and implemented. These tools enable computer assisted design or do some processing of the requirements such as checking, proof, maintenance, and recording on a central data-base.

A system called SSA (Structured System Analysis) [Gan79], [DeM78] assists in structured design and tries to bridge the gap between the requirements and the design of software development. The representation is made by data-flow diagrams that describe the flow of the data among the various processes. Every process is described by a special language called PDL (Program Design Language) that is defined in structured English or by decision tables.

SADT (Structured Analysis and Design Technique) was developed by Douglas Ross [Ros77] [RoS77] as a method of modification of complex systems, and can be utilized on systems other than data-processing. A description is made by model diagrams called *artigrams*. The model presents process, input, output, constrains, and mechanisms and enables hierarchical decomposition of any process into sub-processes.

One system that has relatively wide use is PSL/PSA (Problem Statement Language/Problem Statement Analyzer) [Tei77] developed by Daniel Teichroew from the University of Michigan as part of the project ISDOS (Information System Design and Optimization System). This is a formal language for the statement of problems that is assisted by the analyzer PSA. PSA accepts PSL as input, executes various checks for similarity of input and output, gaps in information flow, etc., and produces a set of reports that describe the system, external and internal data-flow, and more. PSL is based upon definitions of objects and their relation and includes about 20 object types and about 50 relations. The language is quite similar to a natural language and can be learned quickly. It is a very effective documentation tool as well.

At the University if California, Los Angeles, a research was done that relates requirements definition and design methodologies by integrating and enhancing the work of the System ARchitects Apprentice (SARA) Project [Est78] and the ISDOS Project [Win82]. SARA offers support to a designer in creation and analysis of multilevel models. While expressing a need for a requirements definition subsystem SARA has no appropriate requirements definition language, or related tools or methodologies. SARA and PSL/PSA offer some unique strengths but also need each other's strengths.

A large methodology has been developed in TRW, Inc. as a part of the ABM project (U.S. Army Anti-Ballistic Missile Defense), named SDS (Software Development System). A component of the system named SREM (Software Requirements Engineering) [Alf77] [Bel77] deals with requirements. The system was built for dealing with asynchronous real-time systems but can be applied to general interactive systems. It enables description of parallel processing. The processes are described by a language called RSL (Requirements Statement Language). The system has tools that assist in production and checking of the requirements.

At the University of California, San Francisco, a methodology for software development is under development. It has been named USE (User Engineering Methodology) [Was77] and a part of it is the USE Specification Method. This system is being built for the development of interactive systems that have three parts: user interface, a data base, and users' actions on the data base. In this system, transition diagrams are used to specify the user's interface. The data base is specified by using a relational data base scheme, and a variety of notations ranging from natural English to algebraic specification are used to define the semantics of the activities.

Two program design languages that can be used for requirements were developed by Israeli companies. A system called SuperPDL was developed by Advanced Technology, Ltd. [Bur83] [ATL84]. SuperPDL is an interactive system which assists software designers in both the mission level and in the detailed design phases of the software development. It utilizes a non-rigid syntax allowing the designer to express his/her design ideas in a natural language. Another system named SDP was developed by Mayada Ltd [May83]. it has some common characteristics with SuperPDL but is batch-oriented and has no data base.

All the above mentioned systems are more design-oriented, but still can assist in formulating and processing requirements. Generally they are used by the technical staff during system development. Usually it is hard to assume that the users of the systems will understand the requirements that have been formulated in this way.

Many other articles have been published on formal representation of requirements. This kind of formulation – algebraic or logical – is based on formal mathematics in order to achieve precision and conciseness and to enable requirements checking by theorem proving.

SRI International has developed a design system named HDM (Hierarchical Design Methodology). The system under design is specified a TLS (Top Level Specification) written in a language called SPECIAL. There are tools for proving the consistency of a TLS. There are also tools for proving that the TLS in fact implemented by a given high language implementation [Rob76] [Rob77] [Lev79].

A methodology named FDM (Formal Development Methodology) [Kem80] was developed by SDC. Its formal specification language is called Ina Jo®, and it also has tools for formal verification of the consistency of a TLS and of the correctness of an implementation of a TLS.

In the University of Texas at Austin, a language named GYPSY has been developed. It can be used both for specifications and implementation. The system enables development of relatively small programs with high level of reliability. That is done with the help of a proof system called The GYPSY Verification Environment [Goo78]. Another system named AFFIRM [Mus80] [Tho81] was developed at USC-Information Sciences Institute. SLAN-4 [Bei84] was developed as a formal tool for specifying and designing large software systems. It provides language constructs for algebraic and axiomatic specifications and pseudocode for the design phase.

Some articles that discuss the formal approach to requirements can be found in the book *Research Directions in Software Engineering* by Wegner [Lis79] [Lis75] [Gut77] [Hoa73] and in Yeh's book *Current Trends in Programming Methodology* [Yeh77].

All these formal approaches enable a precise definition of a system or a part of it. Their drawbacks lie in the fact that they cannot be understood by the user, and frequently they cannot be understood by the developing staff. Therefore a special person having special training has to deal with this aspect of the requirements. Unfortunately this may have an undesired contribution towards broadening the gap

®Ina Jo is a trademark of SDC, a Burroughes Company.

between users and system developers. Proof of the requirements can be achieved only partially or for relatively small systems. However one must consider the problem and cost of failure. In some cases the problem is tough enough and the cost is high enough that formal techniques though difficult are warranted.

Several commercial software companies have developed detailed methodologies for software development. Pandata B.V. has produced a methodology called SDM (System Development Methodology) [Hic79] that includes a detailed checklist of all the activities that are needed during the life-time of a software project. The book describing this methodology [Hic79] has a chapter that is dedicated to requirements.

Atlantic Software, Inc. has produced a more detailed methodology based on a set of specially designed forms and instructions for filling them as work progresses. The name of the methodology is SDM/70 [SDM70] and its subsystem that deals with requirements is called SES (System External Specifications).

Lately one can notice a tendency towards fitting a specification language to certain types of application with common characteristics. Wasserman's work on USE [Was79] was already mentioned.

Specifications for process-control systems can be defined by a system called ESPRESO [Lud82] [Lud83], which was developed at the Nuclear Research Center, Karlsruhe, Germany. ESPRESO consists of a set of concepts, a specification language, a tool for management, evaluation, and validation of specifications, and a method for the use of the system. The formal approach introduced in [Hei83] produces abstract requirements specifications that apply to a particular class of systems, those that must reconstruct data that have undergone a sequence of transformations.

A family of specification languages that deal with telecommunication applications can be defined using a system developed at GTE [Dav82]. Specification for the broad class of embedded systems are discussed in [Zav82]. The approach is embodied by the language PAISLey. The specifications that are generated are operational in the sense that they are an executable model of the proposed system interacting with its environment.

Surprisingly, only few articles try to point out ways of involving the system user in the process of defining the requirements [Ber83] [Lun79].

It seems that most of the work that has been done so far concentrates on finding ways to represent the requirements and to process them, mainly for verification. There seems to exist a hidden assumption in all of this work of a priori existence of requirements and of a basic knowledge about the desired system and its delineation. That is simply not the case with large software systems. Not all the requirements are known, and in many cases the specified system in not understood. In other words, there is a considerable gap between the fuzzy ideas floating in the users' heads and the more precise requirements statements required by the tools and methodologies described above.

Furthermore, it seems that the users are overlooked in most of the work that has been done so far. Consequently, the requirements are formulated in a way that is not understandable by most of the users and all the validation checks are made among the requirements and not between the requirements and the user's perception of the system.

Therefore, this thesis attempts to develop a method to produce requirements for a large software system. The method should assume no prior knowledge about the requirements and should be user-oriented. Starting from the users' ideas and ending with a precise, albeit natural language, statements of requirements suitable for processing by existing tools. The method should be independent of the way the requirements are presented. Nevertheless, natural language, as the main vehicle of communication should be explored.

1.3 Research Goals

a. To develop a complete and practical method to produce requirements for large software systems.

b. To identify the automation possibilities in this method and to specify an automated system for requirements.

c. To understand better, via this methodology, the notions of system, user, and requirement and how a system is represented.

d. To analyze the role of natural language as a way to represent requirements.

e. To build automated tools for further research.

1.4 Outline of Thesis

Chapter 2 describes and defines a large software system and states some of the difficulties in dealing with such systems. Chapter 3 outlines the proposed method for producing requirements to large software systems. Each stage is fully described and analyzed in the subsequent chapters, and is subdivided into more elementary steps that are detailed in sections. In each chapter, one section is devoted to the automation possibilities of that stage. Chapters 4 to 8 describe the first stages of the requirements production method (4 - Study, 5 - User Identification, 6 - Interviewing, 7 - Raw Requirement Processing). The sequence is interrupted in Chapter 9 to describe a prototyping of automation of the previous stages. CAS (Computer Assisted Specifications) is described and exemplified. The next stages of the method are described in Chapters 10 and 11 (10 - Functional Analysis and 11 - Operational System Specification). Suggested outline for the final products appear in Appendix A.

After the detailed description of the stages, Chapter 12 reviews the automation possibilities of the whole method and suggests an automated system named CASS to deal with this automation. CAS that is discussed previously is a subsystem of CASS. The last Chapter (13) comes to the conclusions and proposes further research in this area.

CHAPTER 2
Large Software Systems

2.1 Introduction

This chapter defines the notion of a large software system, discusses its characteristics and its users, and addresses the difficulties encountered during its life time.

2.2 The User

Speaking informally, the word "users" means all the people that have anything to do with a system, feeding it with data or using its output. These people are supposed to determine the system's characteristics in the initial development phase and during its life cycle when the system is in operation. Other people are supposed to be in charge of operating or maintaining the system without directly benefiting from it and also influence the system requirements.

Computer-embedded systems that usually operate in real time are activated by or invoke external processes. These processes do not result from human activity. For example, an automatic computer-embedded air defense system reacts to signals that come from planes, radars, etc. In this case, the plane or the environment that generates the signals can determine, to some extent, how the system looks. Therefore, a user can be a non-human entity, a part of the outer environment, or a signal generating source.

The trend today is to integrate several individual systems to operate together. Thus, it is possible that a part of an organization is a user of the system and affects its requirements. As an example, in a total banking system, a central exchange system that serves all the banks can be looked upon as a user, even though it is not possible to identify a human individual in it.

Therefore, an *abstract user* of a system is defined to be an entity that affects the systems requirements. An abstract user can be a person, a function, an external process, an organization (or a part of it), another system, etc. Later, it will be seen that an abstract user can be hierarchically decomposed into other simpler abstract users. This is analogous to functional decomposition of systems in the design phase [Par79]. The abstract user is perceived by the system through the requirements that it generates. The system is defined and shaped by these requirements. In order to develop a well-defined system, it is necessary to identify all its users and let every user generate all his, her, or its requirements.

Sometimes it is necessary to go back from the design phase to the requirements phase during system development. This happens when it is found that design problems or limitations change the system's requirements. Therefore, a system's own design can act as an abstract user, or more generally, a system can be a its own abstract user. This recursive relation can be expressed in simpler terms by stating that characteristics of an old system influence the requirements of a new system that replaces it.

2.3 A Large Software System

A *large software system* is defined as a software system that has a large and diversified community of users and entails a variety of human, organizational and automated activities, and various, sometimes conflicting, aspects of different parts of its environments. A description of the behavior and evolution of such systems can be found in [Leh82], [Bel79].

A large software system tends to be physically very large (hundreds of thousands of lines of code, hundreds to thousands of pages of documentation, etc.). However, not all such physically large programs are large software systems as defined above. For example, a program that solves a complicated set of differential equations may be very large, but does not have a diversified community of users, and therefore, does not fit the definition of a large software system.

Many computer embedded systems exhibit the characteristics of large software systems as is mentioned below. Very often they have a very diversified community of users of which some are human and some are abstract users such as external devices, environment, control signal generators, etc.. A description of large, embedded software systems can be found in [Zav82], [Fis78].

2.4 Characteristics of a Large Software System

a. Diversity of users: As mentioned above, this is the main characteristic of large software systems. The users are diversified. Their requirements and the way they perceive the system vary and sometimes even contradict.

b. Multitude of users: This feature is not necessarily a part of the definition. However in reality, the diversity of the users community yields a multitude. The number of users of a large software system ranges from several tens to several thousands.

c. Physical size: A large software system tends to be physically very large. Typically, it has hundreds of thousands of lines of code, hundreds to thousands pages of documentation etc.

d. Resources for development: The development time of a large software system until its initial operation is typically five years or more. The costs rise from hundreds of thousands to tens of millions of dollars. The development staff peaks at tens or hundreds of people. Typically, the size of the specifying team is over ten people.

e. Influence: A large software system generally plays an important role in the operation of an organization or of a major process. Stoppage, disturbance, or a malfunction of the software or its underlying hardware can inflict severe damage to the organization.

f. Cycle life time: A large software system is built for a continuous operation for a long time. Assuming adequate maintenance, 10 to 15 years is a fair guess.

g. Maintenance: A large software system is being developed all throughout its life cycle. Users' requirements change; new users, new requirements, and new applications are introduced, and hardware units change. A large software system is usually dynamic. Previous data shows that the cost of the maintenance from the initial operation to the end of the life cycle time mounts to 3 to 5 times the cost of development until the initial operation.

2.5 The Difficulties in Developing a Large Software System

A large software system possesses all the difficulties encountered in any development of a large scale project: a need to manage and control many persons for a long time and a need of tight control on resources and timetable.

Any software development, whether small or large, has its own drawbacks. Inavailability of proper tools, lack of standards and procedures, unstructured design activities and lack of knowledge, experience, and tools for conducting satisfactory testing and quality assurance.

To these difficulties, others are added that are inherent properties of a large software system. A single person can hardly perceive the system as a whole and analyze it as such. This confronts the project manager and his staff with enormous problems that normally go beyond the analysis phase. Having a diversity of users with all their conflicting requirements and their differing attitudes towards the system (enthusiasm, cooperation, indifference, hostility) brings about the need for compromising and for setting priorities. A total perception of the system, which is sometimes impossible, should be replaced by virtues, such as abstraction, capabilities, and intuition.

The diversity and multiplicity of the users sometimes cause the development to proceed with incomplete or unsatisfactory requirements. This may lead the system to be unusable to a certain degree ranging from lame operation, in which only some functions can be carried out, to total inoperation. There are some techniques today that address the problem of checking the requirements *among themselves*. There is no way yet to check consistency and completeness compared to the *real needs* of the users. Furthermore, it is difficult to identify all the users of a large software system. It is quite common to overlook a user or even a whole group of users.

2.6 Errors in Requirements

An error that is made in the requirements phase of a software project has a more severe impact than an error that occurs in a later stage of development (such as design or programming). A rule of thumb indicates that the earlier the stage in which an error is made, the higher is the cost and the longer is the time needed for correction. This problem is most severe if a mistake is made during the system definition, because such a mistake can affect the overall performance or usability of the system. Considering the resources and the time needed to develop a large software system, it is clear that no effort should be spared in order to achieve good and correct requirements.

Errors in requirements can be classified into several levels (by increasing order of severity):

a. Error in a single requirement.

b. Absence of a single requirement.

c. Inconsistency among some requirements.

d. Inconsistency among groups of requirements.

e. Absence of a group of requirements.

f. Overlooking a single user.

g. Overlooking a group of users.

It is desirable to find a method, or a class of methods, that could ensure completeness and consistency of the requirements as applied to the user and his needs and a method that aids in identifying all the users of the system.

The requirements of a system are likely to change during the system life cycle. The dynamics of the environment will probably force it. The above mentioned methods should allow dealing with changing requirements.

2.7 Cost

As mentioned above, the resources that are allocated for developing of a large software system are quite sizable: hundreds of thousands to tens of millions of dollars during a period of time of five to ten years until the initial operation. The costs are three to five times more for the whole cycle life time. During this period, tens to hundreds of people of various levels and capabilities will be involved in the system development. These people are usually difficult to find and are quite mobile.

The cost of the system, as large as it may be, is minor compared with the costs incurred by inoperation, mal-operation, or late operation of the system. It is very difficult to evaluate in exact figures the damage to a bank of not knowing balances for some days, or to estimate the damage that a country may suffer because of malfunctioning of an air defense system. The harm done may go beyond just money loss and can be described as a major catastrophe. Equally difficult is to quantify the problems caused by inoperation of a system on time. It depends on many factors, such as the type of the organization, the vitality of the application, the existence of a back-up system, etc. Words like "enormous" are usually used as an indication.

2.8 Organizational Problems

There is a perennial shortage of skilled people in all levels of software development. At the more senior technical levels, the lack is more severe and more significant.

Several methods are in common use nowadays that facilitate and help in programming and managing programming teams. There are various possibilities to use higher order languages, application generators, programmer workstations, and other tools that simplify the programmer's work and increase his productivity. Therefore, the overall demand for programmers may ease. New methods and tools for designers, such as structured design and program design languages,are available now. This promises better ways for designing followed by limiting the demand for designers.

The situation is different for the phase of system definition and requirements analysis. The skills desired of the specifiers are not evident yet. It seems that a good specifier would have to have a high technical level of proficiency, a system approach with applications-oriented understanding, and management and psychological knowledge in problem solving and related topics. Hiring and training such employees is extremely difficult. On top of that, their work methods are not structured and no proper tools exist to assist them. The result is that specifying a large software system is a very difficult task. It seems that managing ten specifiers can be tougher than managing fifty programmers.

2.9 Conclusions

The concepts of a *large software system* and of an *abstract user* were defined. The characteristics of a large software system were laid down. The importance of the requirements in the development process and the difficulties in forming them were discussed. Rough estimates of the money, people, and time invested in the implementation of a large software system were presented. The necessity for a smoothly operating system leads to the need for a method to *produce* requirements. This method has to address the problems of completeness and consistency of the requirements in relation with the users' needs and to ensure a full coverage of all the users.

CHAPTER 3
Outline of the Requirements Production Method

3.1 Introduction

This chapter outlines the suggested method for requirements generation and analysis of a large software system. The following chapters detail the stages that are outlined here.

3.2 Purpose

The suggested method attempts to achieve the following goals:

a. provide an organized work plan for requirements analysis,

b. identification of all the users,

c. ensuring completeness and consistency of the requirements,

d. obtaining users' cooperation in the process of requirements generation,

e. increased perception of the large software system by its developers, and

f. computer-assisted processing whenever possible.

3.3 Steps of the Requirements Production Method

1. Study – initial study of the organization and the environment in which the desired system is expected to operate;

2. User Identification – identifying all the potential users of the system;

3. Interviewing – addressing all the system users, and identifying their problems and their requirements;

4. Raw Requirements Processing – processing and organizing the requirements from the previous stage. At this stage, the users confirm and rate their demands;

5. Verification – checking the requirements for completeness and consistency versus the needs of the users;

6. Functional Specification – analyzing the requirements and classifying them by subject;

7. Operational System Specification – identification of basic system operations and of system processes and determining how the system interacts with its environment;

Figure 3-1 graphically shows the outlined process.

Input	Stage	Output
written material (doctrine, operation, procedures, reports), interviews	Study	
	User Identification	users tree
interviews, discussion, written material	Interviewing	interviews summaries, list of problems
requirements	Raw Requirements Processing	list of requirements rated by the users
users tree, requirements	Verification	requirement tree, errors
rated requirements	Functional Specification	subjects, requirements mapped to subjects
functional specifications	Operational System Specification	basic functions,processes

"Figure 3-1: An Outline of the Requirements Production Method"

3.4 Flow of Information in the Process

The basic unit of information in the suggested method is a *requirement*. It undergoes several modes of processing along the various parts of the process. A requirement is generated, rated, classified, refined, composed with other requirements, verified, mapped, and sometimes removed.

Other information items that participate in the process are abstract users and abstract data types. Abstract users are generated at the beginning of the process and are decomposed in order to allow identification of all the users. Abstract data types are usually formed together with the requirements and undergo the same processes. They serve as the logical basis for the data directory of the specified system.

The ability to define basic information units that are processed is an essential step towards automation. A lot of human involvement is necessary here because of the need for unstructured knowledge, processing, reasoning, intuition, decisions, etc.. However since the number of basic information units can be quite considerable (tens to thousands of users, hundreds to thousands of requirements and hundreds to tens of thousands of abstract data types), even the most basic tasks of automation, such as recording, document processing, items control, etc. are of a great help to the system specifiers and to the project management. Further steps in computer-assisted specifications are discussed for each stage separately and an overview is presented in Chapter 12.

3.5 Inputs and Products of the Process

The inputs to the first stages are quite vague and unstructured. Among them are various staff papers, doctrines, written procedures, organizational charts, documentation of other systems, etc. The most important inputs are the ideas of the people that are involved in the system. Sometimes these ideas appear in a contractual form. The purpose of the initial parts of the method is to materialize and structure the inputs in order to allow further processing.

The products of the method are the requirements of the system with attached abstract data types, users list, and outline of implementation alternatives. These products are the base for the design process and the logical nucleus of the system that will be updated during its life cycle.

3.6 Discussion

The main significance of the proposed method lies in the fact that it attempts to convert unstructured activities into an ordered, disciplined process, and to deal uniformly with large numbers of users and requirements.

There is a separation between the field-work, i.e., information gathering and investigation of users, and the analysis work that is done by the specifiers. This approach yields two different levels of requirements and defines a single requirement to be the basic unit of information in this method. By structuring and defining information units, a partial automation of the method is possible which can assist the specifier in his job.

One of the main concerns in specifying a system is to ensure a full coverage of the users and the requirements. Some techniques that address this problem are incorporated into the method. These techniques are useful if they can compare system specifications to the needs of every user or groups of users. Checking specifications among themselves is important because it indicates that there are no contradictions and omissions to a certain level, but the ultimate test remains to be versus the users.

Many parts of the process have to be done as a team effort, allowing the users and the specifier to participate together. Problem solving techniques that are incorporated into the method attempt to enable this togetherness.

It is important, though, to note that the suggested method can neither be fully standardized nor fully automated, since it is essentially a process of design and reasoning. The purpose of the method is to introduce systematic order, to allow more people to carry out specification tasks, and to better utilize the people that participate in the process.

CHAPTER 4
Stage 1: Study

4.1 Introduction

The first stage of the Requirements Production Method (RPM) is getting acquainted with the customer, the organization in which the planned system will operate, the environment, etc. So, this chapter discusses this task briefly.

Learning about the customer and his organization is a continuous process that is carried out during all the requirements gathering phases well into the design phase. It is hard to anticipate and to acquire all the necessary knowledge during the first stage, especially when a large software system is specified.

4.2 Purpose

The purpose of this stage is to acquire basic knowledge about the customer, his organization, the environment where the planned system is supposed to operate, etc. This knowledge is aimed to serve the specifiers during the whole method, but mainly for the next stage: users identification.

4.3 Method

a. An overview of the planned system, goal, should be presented by the customer to the specifying team. The main issues in this overview are: the problems that lead to the system, the system's goals, the environment in which the planned system will operate, organizational structure and functionalities, description of any existing system, automated or not, that the planned system should replace.

b. A review of the formal contract (or any equivalent document) should be written.

c. Any other relevant documents that the specifying team should know should be identified. Such documents are organizational charts, procedures, written doctrines, documentation of automated systems, feasibility studies, etc.

4.4 Automation

During the requirements production method many documents are piled up. These documents are accessed several times in several stages. Therefore, having an on-line the list of documents with brief descriptions, keywords and pointers to physical location, can be of help later. In fact having the documents on-line can only help, as it is then possible, e.g., to use editors to search for occurrences of significant terms, etc.

CHAPTER 5
Stage 2: User Identification

5.1 Introduction

One essential property of a large software system is a multitude of diversified users. Hence, any methodology that deals with such a system has to identify the users first. Then, these users are approached and interviewed, they utter their requirements, and the requirements are further processed. Therefore, a proper identification of users is a cornerstone of this methodology.

Traditionally, methodologies for requirements specifications (e.g., [Ros77], [Alf77]) focus on the system's functions. Their primary concern is what the system does, with only secondary attention to who uses it. The system's users seem to be obvious and therefore, frequently overlooked. Yet, system specifications, which are poor from the users' point of view, may successfully pass consistency and completeness tests with reference only to the system's functions.

The methodological approach that is suggested here focuses on the system's users and looks at the system's functional specifications as the output of processing users' requirements. A system is conceptually represented as the net sum of the users' views (this term is defined later). Figure 5-1 illustrates a system representation by several users (denoted by the letters *a*, *b*, *c*). Only part of the system is covered by the users' views. Adding more users' views will result in a fuller coverage of the system.

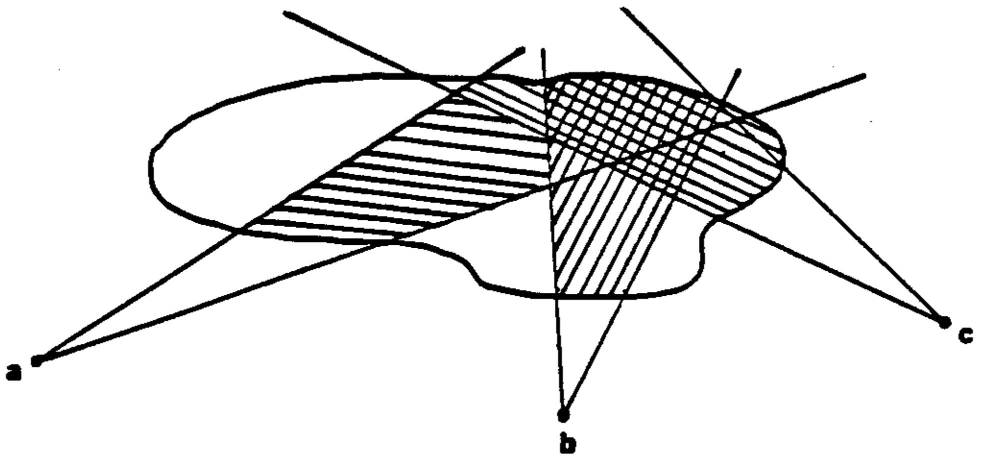

Figure 5-1: A System and Users Views

A user-oriented approach seems to be logically simpler than an approach which starts with an attempt to write functional specifications. A user is a better defined and understood entity than a function. A user can be approached, interviewed, and related to. Therefore, starting with users is more natural to the specifier.

One of the most common and most serious faults in system development is overlooking a user or a group of users. Since a user typically generates several requirements, a significant portion of the specified system may be overlooked. Any mistake in the specification bears severe consequences for the developed system, not to mention the failure to define a part of it. A large software system has many users and is more prone to the danger of overlooking some. Thus, the first task of the specifying team is to identify all the system's users.

A method that attempts to overcome the problems of user identification is suggested in this chapter. The next section explains some of the terms that are used henceforth, and then the method of hierarchical decomposition of abstract users is discussed. Afterwards, the automation possibilities of the suggested process are presented.

5.2 An Example: A Banking System

A banking system is demonstrated through this thesis in order to illustrate some of the concepts of the proposed method. The banking system under consideration should take care of all day to day operations of the branches and their various activities. This includes a variety of services such as ,deposits to and withdraw accounts; loans, mortgages, etc.. Other activities that fall within the scope of the desired system are interfaces with the bank management and in-branch management and control. This kind of a system is indeed large and complex and falls within the definition of a large software system. It is difficult thought to determine at first glance its users. Still, many of its applications are commonly known and do not need lengthy explanations. Figure 5-2 shows part of the user domain of this system. The main groups of system users (to be called later "abstract users") are the branch, the customers, the bank management, and the adjacent automated systems which interface with the designed system. Each of those groups of users are decomposed to its main constituent parts. E.g., the user branch is described to entail the users: branch management, tellers, and back office clerks. Figure 5-2 is self-explanatory, and looking at it reveals how further decomposition is done.

5.3 Abstract Users, Requirements, and Data Types

An *abstract user* of a system is defined to be an entity that affects or can affect the system's requirements and not just an entity that can invoke the system's functions. An abstract user can be a person, a group of persons, a function, an organization or a part of it, an external process, another system, etc.

Under this definition one can identify all the persons that provide inputs or use the outputs of the system, real-time processes that invoke the system or that are invoked by it, the organization that uses the system as a single entity, the system's operators, etc. Other systems may affect this system's requirements through interaction with it. In many cases, the system's design may affect the requirements, usually by imposing design constraints, and therefore the design itself can sometimes be considered an abstract user. Abstract users may be defined at several levels of generalization.

An *abstract requirement* is a property of a system that is perceived by an abstract user. This may be a collection of simpler requirements that can be further decomposed. There exists a many-to-many correspondence between abstract users and abstract requirements. One abstract user can originate several abstract requirements, and an abstract requirement can be originated by several abstract users.

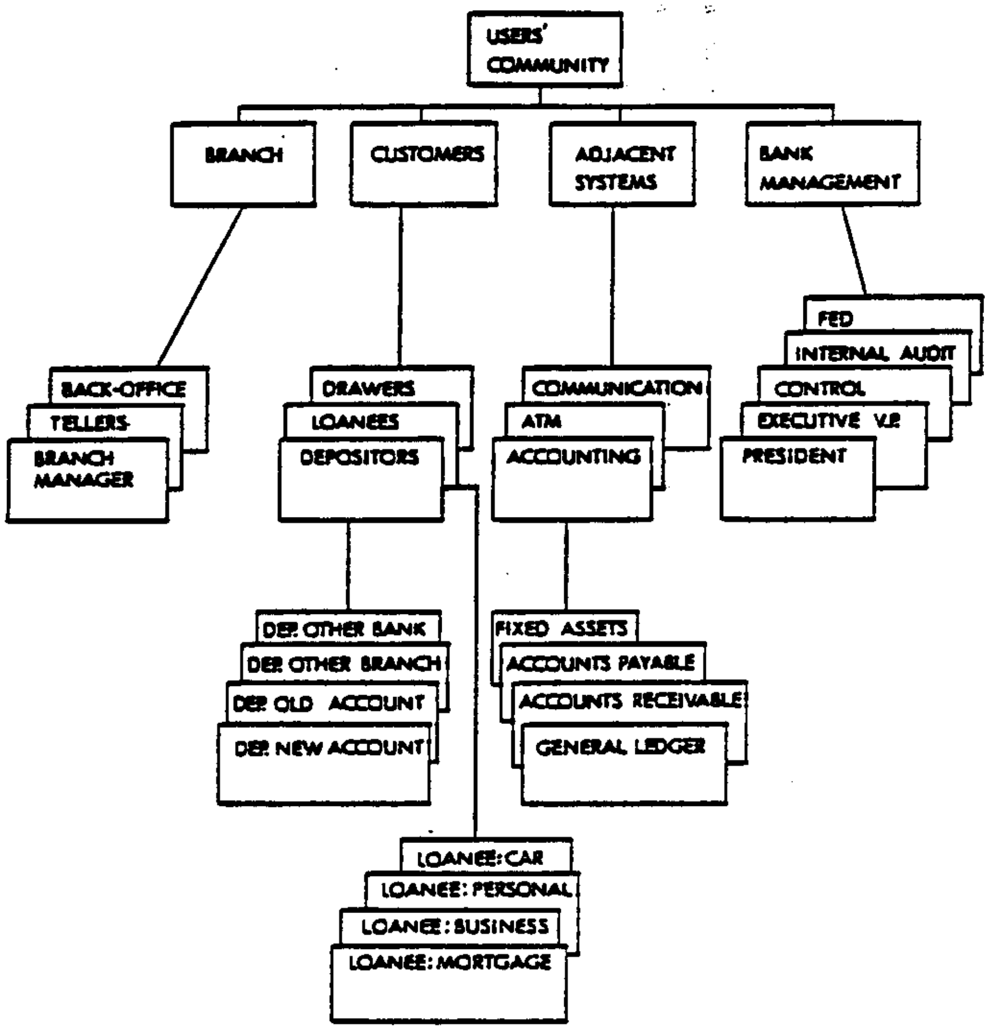

Figure 5-2: Users' Domain of a Banking System

Following the decomposition of abstract users we arrive at a point where the requirements associated with a given user may be fully delineated.

Requirements can often be represented as an imperative sentence with a function as the verb of the sentence and abstract objects as the direct object of the sentence [Kem80]. In this case, it is convenient to use *abstract data types* [Lis75] to group definitions of operators with definitions of their objects throughout the requirements definition. In the forthcoming example, functions and abstract data types are used to specify requirements.

The scheme may thus be summerized as that of an abstract user who generates abstract requirements that are expressed by functions and by abstract data types. A scheme of an abstract users tree with requirements generated at the elementary abstract user level is given in Figure 5-3.

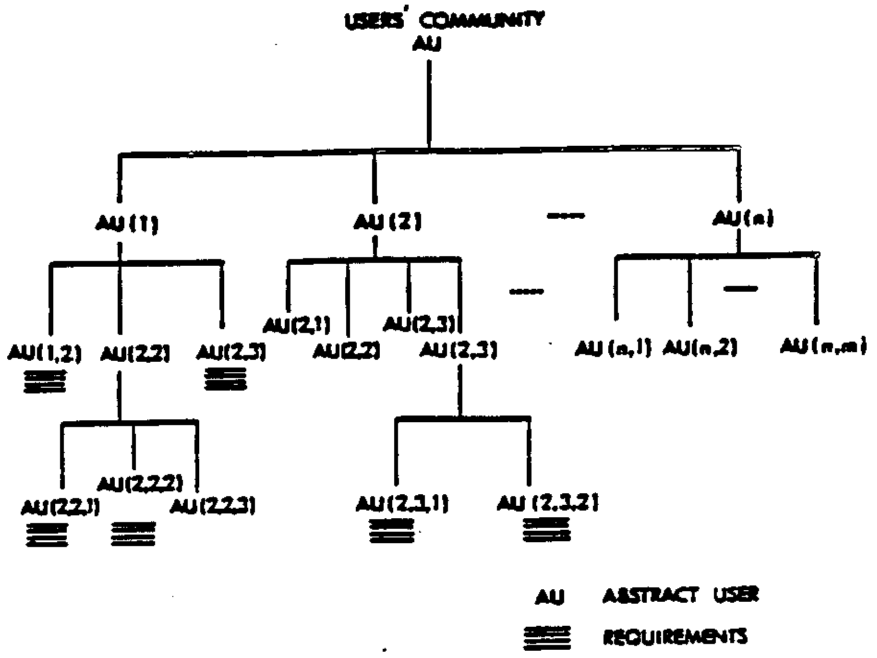

Figure 5-3: An Abstract Users Tree

5.4 Method

1. Determine the highest level abstract user.

2. Decompose every feasible abstract user.

3. Repeat step 2 horizontally and vertically until the abstract users tree is exhausted, and no further decomposition is possible or useful.

4. Investigate every individual elementary abstract user and determine its requirements (See chapter 6).

5.5 Hierarchical Decomposition of an Abstract User

The process of decomposing the abstract users of a system starts with the generalized community of "system users" and is repeated hierarchically up to the level in which further decomposition is no longer possible or useful.

An *elementary abstract user* is an abstract user at the lowest level. These users usually represent a well defined entity such as persons who can be interviewed, processes that can be analyzed, organizational structures that can be investigated, etc. Thus, the process of generating requirements, once the elementary users has been identified, is more natural and simpler to carry out. The trade-off is some redundancy during the process of requirements elicitation, because the same requirement can be originated by several elementary abstract users. This redundancy however, will be shown to be very useful for consistency and completeness checks. It does not pose any burden on subsequent system development since it can be easily eliminated from the final product of requirements specifications. The final result of this decomposition process is an *abstract users tree*, which fully describes the user domain of the system and includes all the abstract users and their decomposition. The next stage – Interviewing (chapter 6) involves generating the requirements of the elementary abstract users.

This method, if carried out thoroughly and carefully, enables the system specifier to arrive at a good coverage of the system users. Its main advantage is in controlling the complexity of the users domain by considering at any given time only the few abstract users that result from one upper level abstract user.

Criteria for this decomposition include the following:

a. The branching factor should be controlled, i.e., the number of abstract users that originate from the decomposition of a single abstract user should be limited (range of 2-6 is reasonable.).

b. The elementary abstract users should be only those who can be directly interrogated for requirements. For a given abstract user, if this is not the case, then further or a different decomposition is required.

c. At any level, an abstract user should be a meaningful and logically sound entity. It should be a well-defined entity that can be related to (Abstractness is by no means meaningless.). It must not be an incomprehensible gathering of other abstract users that have no common meaning.

Similar criteria for functional decomposition of code can be found elsewhere [Par79].

5.6 Automation

The functions of this stage that can be automated are:

a. construction and maintenance of the users tree.

b. decomposition of abstract users, and

c. managing the specification project.

The entire users domain should be recorded and maintained. User information such as identification, description, links to parents and children must be kept. Further data items such as requirements are added during the next stages.

The process of users decomposition is a human activity that requires knowledge of the specifications domain and a background of similar applications. Here the specifier can be assisted by an expert system that utilizes a knowledge base to draw the specifier's attention to possible pattern of decomposition. The knowledge base results from accumulation of expertise of similar applications. Thus, an expert system can artificially augment the specifier's experience.

Management aids are automated tools that help managing and controlling the process. During the users identification stage, the main management issues are planning and monitoring the tasks and schedules of the specifying team.

5.7 Conclusions

This chapter describes how to identify the users of a specified system to be specified. It expands the intuitive notion of a user to the notion of an abstract user. A process of identifying users by hierarchical decomposition of abstract users is suggested. It is simple and orderly, thus reducing the chances to overlook a user or a group of users.

The notion of a user is basic to the whole specification methodology. The user is looked upon as the source of the requirements, and the specified system as the sum of its users' views.

The next chapters suggest how to approach and interview the users, how to convert interview summaries into requirements and how to check them. The notion of a requirement is further expanded to the notion of an abstract requirement, enabling different levels of user views and leading to a way to check and verify the requirements.

CHAPTER 6
Stage 3: Interviewing

6.1 Introduction

In this chapter, the method of interviewing the user of the system is discussed. In the previous chapter, an abstract user is defined to be any entity that can affect the system's requirement. After decomposing the abstract users and arriving at the elementary abstract users, a process of interviewing, investigation, and interrogation must take place.

An elementary abstract user is, or is associated with, a real, physical entity, that would be usually referred to simply as a "user". She, he, or it is a person, a small well-defined group of people, a process, or other automated system that can be directly approached.

This investigation usually materializes in the form of people-to-people discussion. However well-defined and well-documented the user may be, it seems that this encounter is unavoidable. Even if a certain elementary abstract user turns out to be a non-human entity (e.g., another automated system) still the best, and maybe the only way, to elicit its requirements is by talking to people who run that system or are in charge of it.

Every practitioner and every veteran of the area called "system analysis", anybody who has ever tried to specify systems requirements, has probably experienced the difficulties and frustration of this encounter. Two main reasons are supposed to cause these difficulties. One is the mental and technical gap between the user and the specifier. The second lies in the fact that system's requirements are essentially the expectations of its users, and expectations tend to change. Therefore, a system seems never to be fully specified [Leh82].

This chapter discusses more deeply those difficulties, tries to lay down an orderly method for requirements eliciting from the elementary abstract user, and a way of recording the results for the sake of the next steps in the method at large. This chapter addresses mainly the problem of person-to-person interface, which is a most difficult one to structure. Some techniques that are based on psychological or management techniques have therefore been suggested.

The importance of this stage is evident. This part of the method is where the first connection is established between the system implementors and the system user, where the specifying work starts to become a down to earth task, where computer science encounters reality. Here the cornerstone of the whole system is formed. The new requirements that are generated at this stage are processed, checked, and composed later on to form the base for the systems delineation.

6.2 Problems

Quite frequently, systems tend to turn out differently from what either the users or the designer had in mind. Two main problems have been identified and mentioned above. They deserve a further discussion.

a. *Communication Gap:* The system specifier and the system user are generally persons that differ in many aspects. The typical system specifier is a computer or system professional, whose main interest lies in computers. He speaks the professional jargon (computerese) and is out of the daily routine of the process that he tries to automate. Sometimes he expresses impatience and in extreme cases despise towards the user. The system user (or potential user) is application-oriented and has his own jargon (be it banking, apparel industry, military command and control, or whatever). He knows very well his daily routine and is generally suspicious towards newcomers that suggest changing everything after a discussion of a few hour or a few days.

   This gap narrows if the two sides share the same jargon, but it seems to never fully disappear. Having computer professionals on both sides would not eliminate it either.

   In extreme cases, this relationship develops to plain hostility and sometimes causes the termination of the whole project. Various solutions are designed for such cases, notably adding a third party, a mediator, between the specifier and the user. Rarely is this solution satisfactory. The mediator (whatever his title may be) is forced either to take a side or to form vague results, vague enough to satisfy both sides and vague enough to prevent construction of a well-defined system.

   The mediator approach is institutionalized and established as a part of the organization under different names, such as vice-president for MIS, system analysis department, information engineering or external consultants. Still the experience gained indicates that the mediation approach cannot be the ultimate solution to the gap problem.

b. *Expectations:* The final requirements of a software system should evolve as a compromise among the users' desires and various constraints, such as technical limitations, budget and time limitations, etc. However, whereas the constraints are well-defined and usually intact for a long time, the expectations of the users tend to change and vary, and are vague and volatile. An old saw states that if a system works, it does not fulfill the wishes of its users, whose expectations have already been changed.

   New requirements arise as the environment changes, as new people come along, or as new ideas are introduced. As time proceeds, users' perceptions about automation and about their own system refines, and they find better ways to express their ideas. This yields a stream of new requirements and desired changes to the old requirements.

   At the time when the new system is operational and introduced as such for the first time, users typically are unsatisfied because either they do not see some of their requirements materialize the way they would like them to, or their level of expectation has changed and therefore new requirements are introduced.

   This problem has been addressed so far by a mixture of administrative and technical means such as freezing requirements, configuration management, etc. In other words, at the moment when the user ceases to be indifferent or hostile to the notion of the new system, at the moment when the user starts to be active, excited, intrigued, creative, and wishes to contribute, he has to tackle what seems to him new obstacles. This can inflict upon him a negative attitude towards the suggested system and its specifiers, or broaden the communication gap between him and the implementors.

6.3 A General Approach

The general approach that is suggested here is to look at the requirements generation at the elementary user level as a problem solving process that is carried out cooperatively by the users and the developers of the system. Numerous techniques have been developed to address problem solving issues [Van81]. Some of them can be adopted in requirements generation.

The most important issue here is that of team work. Both users and designers team together to solve a problem. That solution will be implemented in the system and is described by the system requirements. If this approached of togetherness prevails, then the communication gap narrows or disappears, yielding better working relations and better results. The techniques that are described later on are used to create and enhance this form of team work.

One of the best known models for general problem solving process is Simon's [Sim77] three stage process of intelligence, design, and choice. In the first (intelligence) stage, the problem is recognized and information gathered; in the second (design) stage, problem solutions are developed; and in the third (choice) stage, the solution alternative is selected. The process of requirements generation can be looked upon as a problem solving process. The user has problems to be solved by the desired system. These problems have to be identified, information has to be gathered, several approaches to solve them have to be discussed, and finally, a solution has to be selected. Therefore, the requirements eliciting method is a multi-problem (because a single user may have several problems to be addressed) solving method.

Assuming for a moment that the communication gap can be narrowed by using a problem solving technique, still the second problem of expectation remains. No solution to this issue exists, and it is doubtful if a solution is desired at all. Creativity should be looked at as a virtue that has to be encouraged, and a reasonable method has to pave a way to redirect and use the users' expectations rather than to discourage them. A fairly practical assumption is that every system is due to be changed and there is no way to avoid new requirements being introduced.

6.4 Requirements Generation at the Elementary User Level

The process is outlined as follows:

a. Preparatory Stage:

1. information gathering,
2. studying the environment,
3. planning the investigation,
4. preparing of essential questions

b. Investigation Stage:

1. forming a discussion pattern,

2. forming an investigation setting,

3. investigation,

4. preparing minutes of the meeting

c. Documentation:

1. investigation summary,

2. team review,

3. write up of requirements and problems,

4. summary structure

The last stage of a problem solving method, the selection process, is not presented here. It is deferred to stage 4 (chapter 7) in the requirements production method, when the requirements are evaluated, rated, and selected. The reason for this delay is that this stage addresses the elementary individual user. A process of gathering requirements and problems from the other, non-elementary users has to take place before proceeding to a selection method.

In the remaining sections of this chapter, sub-stages a, b, and c are further elaborated.

6.5 The Preparatory Sub-Stage

This stage is performed before going to the specified user that is the subject of the interview. First, information is gathered about his job definition, his place in the organization, main duties, processes, and activities that he is in charge of or participates in, and anything else that can bring the specifier to be more knowledgeable about the user. If the elementary abstract user is a process, a system, or any other non-human entity, then the persons that are most knowledgeable about it have to be identified and addressed.

A thorough study of the user takes place before the interview starts. It is advisable that the user be informed about the specifiers' intentions and activities at this point and recommend to the specifier what kind of information to gather. Then a set of questions are prepared and sent to the user before the interview is done. In this way, he can get himself prepared for the main issues of the interview. Further questions will probably arise during the interview.

The whole process of investigation has to be planned ahead by the specification team. Care has to be taken for items such as time and place, duration, allocated personnel, contents, etc. Under no circumstances should the time be limited, but one session should be limited to two to three hours. Provisions have to be made for further interviews if necessary.

6.6 The Interview Sub-Stage

This stage should be and should look like a team effort rather than a court inquiry. Both the user and the specifier must have the feeling of participating in a common effort. Therefore, the word "conference", rather than "investigation" fits better here.

The conference should include the user and a small team of specifiers. The number depends on the project size and on the needs but is no more than three or four persons. The user can involve other people in the conference, but one session should be dedicated to each single user, in order to make the user feel important enough to warrant a team of specifiers, and to concentrate on this user's special needs.

One person will lead the discussion and another person will be in charge of recording and documentation. A good leader directs the conference, prevents it from going astray, and ensures the full participation of all the attendants. He does not try to impose his own idea or preconceptions (if he has any) on the other participants.

Generally speaking, the setting of the conference will be that of classical brainstorming [Osb53]. This is probably the oldest and best known "creativity technique", in which several people are encouraged to work together on a problem. The group is encouraged to generate ideas and to record them. All ideas are acceptable no matter how outrageous they might seem, and ideas of others should be used to spark new ideas. No criticism or judgement of ideas is permitted until all ideas have been laid down.

The conference starts with an introduction by the conference leader that states the purpose of the meeting. Time is allowed for questions concerning the background material and then an input-output technique is used to direct the requirements generation. The user, followed by other team members, establishes the desired output, the major inputs affecting the output, and any limiting specifications that the output must meet. (This method was first developed by General Electric Company [Whi58] but has been extensively used in various system analysis and software processes. A further development of the method for modern software system was implemented using the desired contents of the screen as the output [Bas81]).

Wishful thinking and outrageous ideas are encouraged. No technical matters or technical jargon that excludes any of the participants should be allowed. The conversation should use application terms rather than implementation terms. Everything is laid out on the board in front of everybody and re-evaluation and feedback is encouraged. No criticism of the suggested ideas should be allowed. However, analyzing existing processes and systems, in a critical way should be encouraged (no personal offenses, of course). A good leader (conference director) takes care of all this.

As a problem solving process, this stage blends several techniques known as Wishful Thinking, Input-Output, Attribute Listing, Classical Brainstorming, and Lateral Thinking (A good description of numerous problem solving techniques are found in [Van81].). The main idea is to allow teams to work in an organized, orderly manner to enhance creativity and idea generation. The team must, in a relatively short time, find the main problems of the user (actually of all the users of the whole organization) in viewing a specified system and to suggest solutions (Note that a solution here is defined to be a requirement of the system.).

The method known as Lateral Thinking deserves a special discussion. It was developed by Edward de Bono [Bon70] to provide a method for escaping from conventional ways of looking at a problem, and for developing new methods and new attitudes to apply to the thinking process. The purpose of Lateral Thinking is to disrupt the normal patterns of thinking by introducing discontinuity and thus enabling new ideas to be formed. The techniques used for this purpose are awareness, suggesting alternatives, and provocation. The requirements analysis phase of systems development is the time to introduce new ideas.

In order to prevent automation of unnecessary processes, or developing new systems that differ from their predecessors only by minor technical aspects, new ways of thinking and generating ideas have to be introduced.

Consequently, good training in problem solving techniques is required. System designers or specifiers, whose backgrounds are typically computer science, engineering, or management, should exercise those new capabilities in order to excel in that task.

6.7 The Documentation Sub-Stage

The results of the investigation stage should be properly documented and used as an input to the next stages. Since several teams are investigating many elementary users, a uniform way of documentation is critical.

As mentioned above, one of the team members is in charge of recording and documenting. However, each member of the team has to feel responsibility for the results and should do his best in influencing it. This should be done in simple plain language and by using easy-to-understand charts (e.g., HIPO charts [HIP74]). No formal presentation of requirements is indicated at this point.

Several papers are distributed among the team members during the first two stages, such as background information, procedures or doctrines of the organization, minutes of conferences, etc. They should be stored for later use and reference by the system implementors, or for use in a case when changes to a requirement are needed.

At the end of the investigation stage, a document called the investigation summary is generated. Its contents are as follows:

a. General information – place and date of conference, names of participants, reference number, name of the abstract user, etc.,

b. Organization chart – the function of the specified user in the organization,

c. Functional description – a list of the main activities and responsibilities of the user,

d. Detailed activities and processes – detailed description of activities and processes in which the user participates, or that are under his responsibility, or result from his activities. The description is laid down verbally or graphically in an easy-to-read and understandable way,

e. Other issues – subjects, activities and processes that do not fit paragraph (d) but have been raised during the investigation,

f. List of problems – a list of all the problems perceived by the team in the existing system from the point of view of the user,

g. List of requirements – a list of all the requirements raised by the team.

Problems and requirements should be laid down in a logical, orderly manner. They should be ordered by subjects (and not chronologically as discussed by the team). No subject that has been discussed is allowed to be eliminated even if it seems to be not important to the editor. This investigation summary document should be reviewed by all the team members before its final issue.

6.8 Automation

Only some parts of the method that is suggested in this chapter can be automated or computer-assisted. It is impossible to conduct a computer-assisted interview. An attempt to do so can frighten and deter some of the users. However, it is possible to take advantage of the computer to edit and to store the interview summary documents.

As discussed in chapter 3, it is highly desirable to automate or to be assisted by computer as much as possible throughout this process. The interview summary documents are input to the next step that forms the raw requirements. Capturing them in the computer at that early phase enables to check for completeness of all the interviews and gives the project manager an important tool for follow-up and for monitoring the project. Word processing capabilities facilitate the editing and the generation of nice documents.

A full survey of automation of the requirements production method for a large software system in given in chapter 12.

CHAPTER 7
Stage 4: Raw Requirements Processing

7.1 Introduction

The previous stage has left the specifying team with a bulk of investigation summaries. This stage incorporates into these summaries other requirements that may appear implicitly. Then, all the problems are converted into requirements and all the requirements are grouped into subjects (classified) and rated by the users.

The purpose of this stage, raw requirements processing, is to form a presentation of the specified system by subject, thus allowing a subject to be perceived and dealt with as a whole. The output of this stage are the raw requirements, requirements that are rated by the users and classified by subjects. They are further verified and processed in the next stages, the stages of verification and functional specification.

This chapter describes the suggested steps of the raw requirements processing stage. Substantial automation can be used at this stage, and the issues that can be subject to automation are discussed at the end of the chapter.

7.2 Method

The raw requirements stage involves

1. incorporation of implicit requirements,
2. preparation of keywords /subjects list,
3. classification of problems /requirements,
4. users' rating of problems /requirements,
5. translation of problems into requirements, and
6. further investigations whenever necessary.

Figure 7-1 describes the outlined process together with the data and the knowledge sources that are involved.

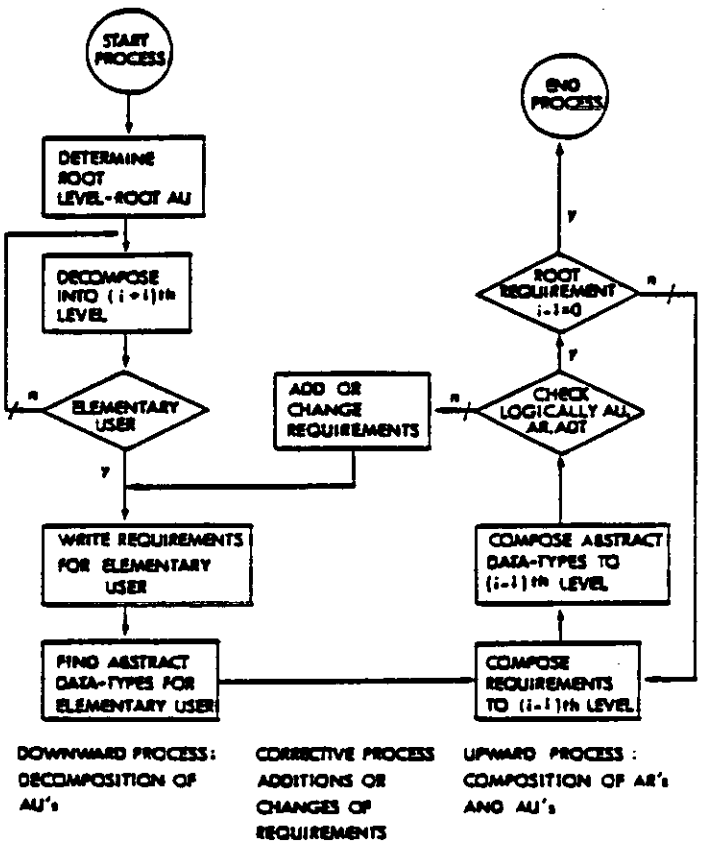
START PROCESS
DETERMINE ROOT LEVEL-ROOT AU
DECOMPOSE INTO (i+1)th LEVEL
ELEMENTARY USER
n
y
WRITE REQUIREMENTS FOR ELEMENTARY USER
FIND ABSTRACT DATA-TYPES FOR ELEMENTARY USER
ADD OR CHANGE REQUIREMENTS
END PROCESS
y
ROOT REQUIREMENT i-1=0
n
y
CHECK LOGICALLY AU, AR, ADT
n
COMPOSE ABSTRACT DATA-TYPES TO (i-1)th LEVEL
COMPOSE REQUIREMENTS TO (i-1)th LEVEL
DOWNWARD PROCESS: DECOMPOSITION OF AU's
CORRECTIVE PROCESS ADDITIONS OR CHANGES OF REQUIREMENTS
UPWARD PROCESS: COMPOSITION OF AR's AND AU's

Figure 7-1: Outline of the Process

7.3 Incorporation of Implicit Requirements

Until now, all the requirements presented to the specifiers are those that are generated by well identified users. Sometimes requirements are generated implicitly by documents, management decisions, etc. These documents should be treated as if they were interview summaries of unknown users. Usually a user can, and should, be attached to these requirements, turning them into explicit requirements. These requirements have to be run through the previous stage and thus incorporated into the explicit requirements. Figure 7-2 shows this sub-process

Sometimes the raw requirements processing process has to be reiterated because of user reviews and comments. These iterations generate more raw requirements, that are classified, attached to users and made ready for the functional specification stage.

7.4 Preparation of Keywords and Subjects Lists

The first step that has to be taken before the classification is initiated is to decide upon the classes. This task is rather difficult because the system is as yet unknown or partly unknown, and therefore the classes into which the requirements will be classified are unknown. It is large enough and undefined enough to make predetermination of classes quite difficult.

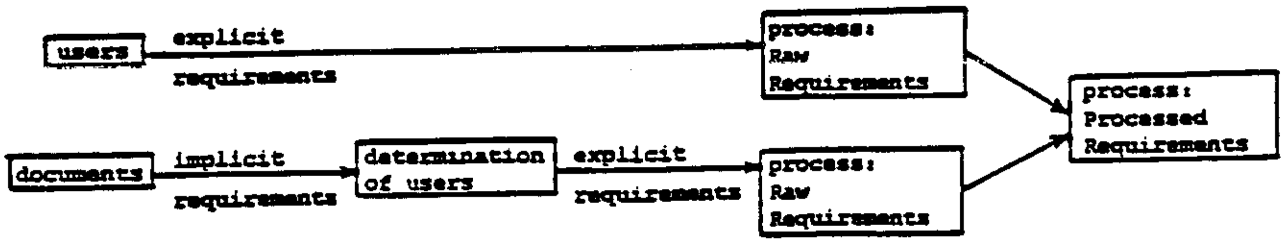

Figure 7-2: Incorporation of Implicit Requirements

Three ways are suggested for determining the classes. They can, and should, be used simultaneously for best results. The three ways are called preconception, statistical analysis, and accumulated knowledge.

a. Preconception: It is unwise to assume that the specifiers do not have any idea about the desired system at this point of time. Gathering data, learning the organization, and interviewing users probably yields some ideas or preconception about the system and its main topics. For example, dealing with a banking system can yield thoughts about such subjects as deposits, loans, accounting control, finance, etc. Therefore, a preliminary list of subjects can be laid down by the specifiers and the users before any processing takes place.

b. Statistical Analysis: A lot of written material about the system was gathered in the form of interview summaries. It is desirable that all of this text is stored on-line and can be processed by a computer. An analysis of the relative frequency of each word can now be done. Words that are more frequent than the others are good candidates to form classes or to lead to classes. Some caution has to be taken though. Commonly used words that are irrelevant such as pronouns and prepositions should be eliminated from the analysis. Secondly, words that have the same syntactic root should be grouped and considered as one word. Skimming techniques are discussed in [Cul78] and [DeJ79].

c. Accumulated knowledge: Previous specifications of other systems that have similar characteristics or operate at similar organization, generated knowledge relevant to the current system. Therefore, use of other subject lists is recommended. This is specially important when a new generation of the same application or a broader integration of several existing systems is planned.

To enable the use of prior knowledge, such knowledge has to be recorded. An application glossary is suggested here to be a list of keywords that are related to an application or a class of applications. A central database of application glossaries can be built in order to assist new specifications. As a matter of fact, one of the product of any new design should be an amended application glossary for future design efforts. This application glossary can be used in other stages of the process such as requirements composition (see chapter 8 and [Bur84]).

Utilizing accumulated knowledge has broader application than it seems in this chapter. Much has been said about reusable software. Reusable requirements are the indicators to the possibilities to use previously written software. Accumulating knowledge is a difficult task to define. Recording keywords is a tiny start in this direction.

It is strongly suggested that all three methods, preconception, statistical analysis, and accumulated knowledge be used. Each of these methods has some deficiencies that can be eliminated by a combined use. Preconception is sometimes incomplete and tends to propagate old or incomplete approaches, thus excluding consideration of new, complete ideas. Statistical analysis is cumbersome (although programs that do this analysis do exist) and when done may be incomplete. For using accumulated knowledge there is a need for similar systems that have been previously analyzed and recorded. A knowledge base of this kind rarely exists, and even if it does, it may not have all the necessary subjects (keywords) for the desired system. On the other hand, each of these methods has its own merits, and, when combined, they can be of a great help to the specifiers.

7.5 Classification of Problems and Requirements

At this stage, problems and requirements are assigned to the classes that have been identified by the previous stage. Problems and requirements are sometimes interchangeable, e.g. one user may state a need for a report as a requirement while another user may indicate the lack of such a report as a problem. On the other hand, solving a problem may lead to a new requirement or a new set of requirements. Resolution of problems and requirements is done in the functional specification stage (chapter 9).

The presently considered classification yields an orderly presentation of problems and requirements, eliminates partly the redundancy in a requirement that has been stated by more than one user, and allows contradicting requirements that have been issued for the same subject. Furthermore, following the classification the system begins to take a shape. Both the specifiers and the users can perceive, for the first time, the system's form.

A large software system usually contains hundreds of problems/ requirements, so the classification can be very long and cumbersome. Some work can be saved by preliminary classification to be done during the interviews and by partial automation. The classification cannot be done entirely during the interviews stage because the classes are not fully known yet.

7.6 Users' Ratings of Problems and Requirements

The interview summaries that are prepared during the previous stage are now transferred back to the users for review and rating. Every user should get the classified requirements that have been generated from what he said during the interviews together with the interview summaries. Generally, this document includes the subjects, the problems and the requirements that are mapped to those subjects, and a list of users that have raised these problem or requirements.

Each user, after evaluating the document, should rate every problem or requirement according to a scale that is supplied beforehand by the system specifier. This scale indicates the importance of a specific problem or requirement to the user (e.g. very important, low priority, unacceptable, irrelevant, etc.). Whenever scaling seems not to be sufficient to express the user's view then comments can be inserted.

The user is encouraged at this stage to add new subjects and new requirements that have not been investigated before or to change his previous requirements. All this new information should be processed again, until it is satisfactorily incorporated with the initial documents.

7.7 Translation of Problems into Requirements

Now the specifier together with the relevant users should go over the subjects and translate the problems into requirements. In most cases, this is possible and can be done quite simply by mere stylistic changes (E.g., the problem: "I do not get a daily balance" can be turned to the requirement: "The system should prepare a daily balance report".). Sometimes problems cannot be easily translated into requirements, and a more thorough analysis is needed. If possible, the analysis should be done at this stage. If it is not possible at this time, it could be deferred to the processed requirements phase.

A natural language processing system can be used here to help rephrase problems as requirements and to help filter the more complicated problems to further analysis by the specifier.

7.8 Automation

There are several possibilities for automating steps of this stage. These are summarized below.

a. Application glossary: One of the methods used to prepare an application glossary is by statistical analysis of the interrogation summaries. Frequent words or phrases can, under some constraints, be assumed to be keywords or subject titles. Assuming that all the interview summaries have been entered on-line in order to take advantage of the computer as a word processor, then searching for repeated phrases is a simple task.

A similar process can assist in the accumulated knowledge process. There we try to match keywords from other systems, that are known to have some similarity to our system, with the above mentioned interrogation reports.

b. Classification: Once a list of subjects has been obtained, then the entire set of interview summaries can be scanned to search for occurrences of those keywords, yielding automatic classification. The unclassifiable requirements should be processed by a human specifier.

c. Management: During this stage, a lot of attention has to be devoted to keeping track of all the data items in the process, such as subjects, problems and requirements, users, etc. The project manager has to assign those data items to persons on his staff and to see to it that each data item has been properly addressed and processed. Computer programs can support all these activities, and help in other management activities such as budgeting, control, and scheduling.

7.9 Conclusions

This chapter discusses ways for the classification of a large amount and variety of requirements and problems, even when the nature of the classes is not fully known a priori. The classification task is important as the first step towards arriving at a system out of discrete requirements. A part of the classification work can be computer-assisted or be entirely automated based on some text-processing tools. The task of converting problems into requirements can be partly automated.

CHAPTER 8
Stage 5: Verification

8.1 Introduction

In the previous stages, the elementary abstract users were interviewed and requirements were extracted out of interview summaries. Naturally, the next stages should further process these requirements, and turn them into specifications. However, before doing so, the requirements have to be verified. The method of verification has to ensure that no requirements are missing and that the stated requirements are correct.

Substantial research has been done on requirements verification (e.g., [Rob76], [Dav79], [Kem80], [Zan82]). Most of the work that has been done so far attempts to represent the requirements in a formal way, which allows algorithmic checking by computer. These checks usually verify the requirements as consistent with each other. This verification is important but is not the last word. A more important verification occurs when the users are confronted with the requirements and find requirements that do not conform with what they desire or find that some requirements or users are missing.

The way that this verification is done is by introducing some redundancy into the process. As was mentioned before, the requirements that are produced so far are elementary requirements that result from elementary abstract users. Requirements can be produced for higher level users, so that several presentations of the specified system result.

This chapter discusses the idea of composition of requirements and its relation to a presentation of the specified system. Then a method of composition and verification is presented. This method is prototyped and demonstrated later (chapter 9).

8.2 The Method

a. Starting from the elementary abstract users (the leaves of the abstract user tree) compose the requirements for the higher level abstract users. Figure 8-1 illustrates the abstract users tree and the order of composition.

b. At each abstract user node, check the composed abstract requirements if they are logically sound. In other words, compare the composed requirements to those that would have been generated by the abstract user.

c. If a discrepancy is found, then either the composition process or the lower level requirements have to be checked.

d. If the problem is found to be in the lower level requirement, then a correction is done: Requirements are modified or user decomposition is done. Thus, this process yields the necessary changes in the users domain and in the requirements.

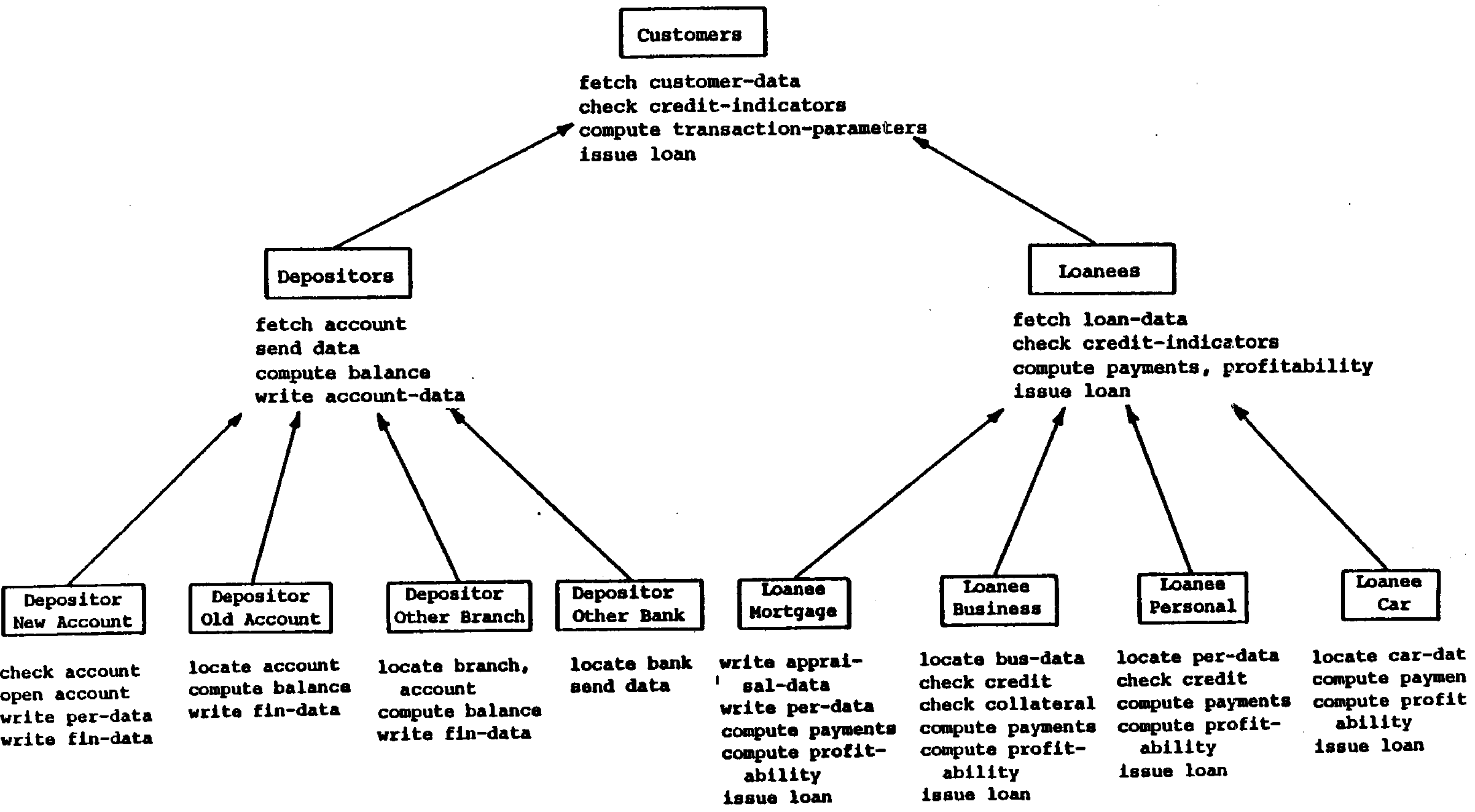

Figure 8-1: User Tree and Requirements Tree of a Banking System

e. The cyclic processes of user decomposition, requirements generation and requirements composition are iterated until no further discrepancies are found.

Figure 8-2 illustrates the method by presenting its flowchart.

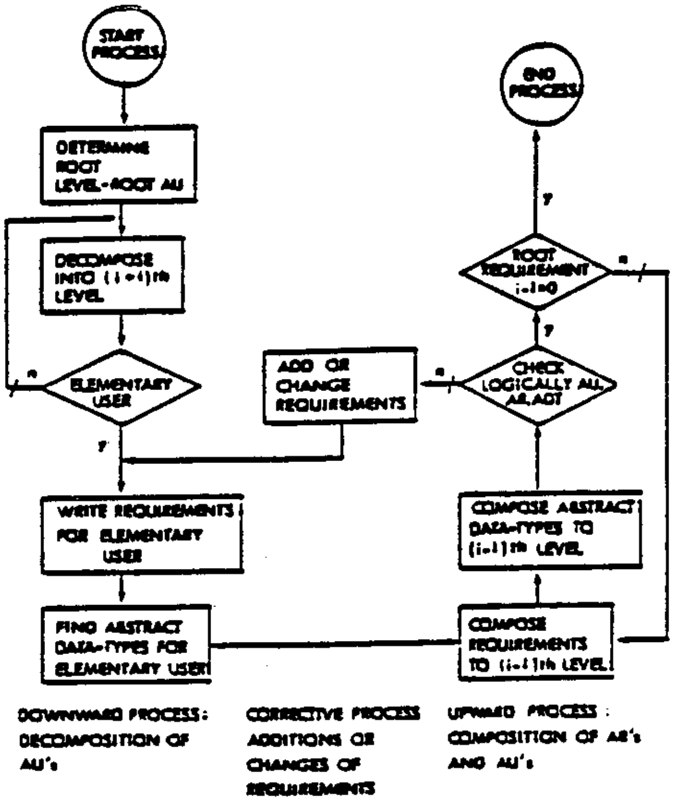
START PROCESS
END PROCESS
DETERMINE ROOT LEVEL-ROOT AU
DECOMPOSE INTO (i+1)th LEVEL
ELEMENTARY USER
n
y
ADD OR CHANGE REQUIREMENTS
CHECK LOGICALLY AU, AR, ADT
ROOT REQUIREMENT i-1=0
WRITE REQUIREMENTS FOR ELEMENTARY USER
FIND ABSTRACT DATA-TYPES FOR ELEMENTARY USER
COMPOSE ABSTRACT DATA-TYPES TO (i-1)th LEVEL
COMPOSE REQUIREMENTS TO (i-1)th LEVEL
DOWNWARD PROCESS: DECOMPOSITION OF AU's
CORRECTIVE PROCESS ADDITIONS OR CHANGES OF REQUIREMENTS
UPWARD PROCESS: COMPOSITION OF AR's AND AU's

Figure 8-2: An Outline of the Users Identification and Verification Stages

8.3 An Example: A Banking System

Before going into formal definitions and detailed discussions, an example will serve to clarify and to illustrate the main issues. The Banking system from the previous chapters is used here. Figure 8-1 shows the abstract users tree combined with the abstract requirements tree. The arrows indicate the direction of the upward composition of the requirements.

Requirements that have been generated at the lowest level of the abstract users such as "Depositors to New Account" (Dep-New-Account), "Depositors to Old Account" (Dep-Old-Account), etc. are being composed into abstract requirements for the abstract user "Depositor". A similar process composes the abstract requirements of the abstract users "Loanee-Mortgage", "Loanee-Business", etc. to the requirements of the abstract user "Loanees". Composing up one level further yields the requirements of the abstract user "Customers" that results from those of "Depositors", "Loanees", and several other abstract user that are not mentioned in the example.

8.4 Bottom-up Hierarchical Composition of Requirements

Composition of requirements is the formation of higher level requirements for higher level users or the *abstraction of requirements*. A *user's view* is the way that the user perceives the system, or all the requirements of the user that pertains to the specified system. Here, this definition is widened to accommodate any abstract user, not only elementary abstract users. Thus, different coverages by users' views exist for the same system. These views should conform if all the users are identified and all the requirements are properly stated. Figure 8-3 illustrates the users views' as cones whose bases compose the system. Higher cones that contain smaller cones belong to higher level users. The top node represents the view of the entire users community.

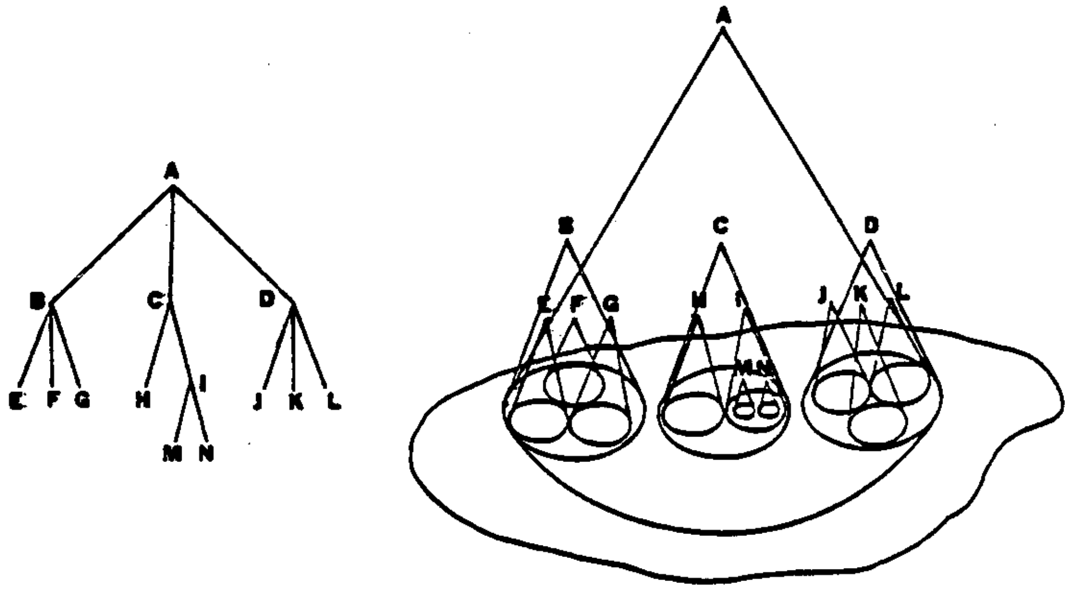

Figure 8-3: Various Levels of Users Views

The essence of upwards composition of requirements is to reduce the total number of requirements and to eliminate details that are unnecessary for that level of abstraction. This is the reverse method of refinement or decomposition. Thus, a set of abstract requirements of an abstract user should represent a less detailed grouping of the combined sets of requirements of its children.

By a similar method, an abstract data type tree can be formed. During the composition process unnecessary details are eliminated and only the information that is relevant to each level of abstraction is preserved.

The motive for the upwards composition of requirements and of abstract data types is to ensure completeness and consistency of the users and requirements and to identify redundancy. Having a set of abstract requirements and abstract data types for each abstract user, the specifier can look at each abstract user, of every level, and check if all its requirements are logically sound. If not, appropriate correction can take place, possibly followed by further decomposition. It is suggested that these checks be performed after forming the abstract requirements and abstract data types that belong to the specified abstract user. Thus, any needed correction can take place before the composition process is completed, thus avoiding unnecessary labor.

The proposed method is independent of the way in which the requirements are presented. Two different ways of presenting the same requirements and their composition are demonstrated in Appendix C. The most natural way for presenting the requirements is to write them down in a free-format natural language. Since it is desirable to provide the specifier with automatic tools, it is suggested that a more formal or structured presentation be used.

Each requirement can be written as an imperative sentence with a function as the verb of the sentence and abstract object as the direct object of the sentence. (The class that the objects belongs to turns out to be an abstract data type at higher levels of abstraction.). The function and the object are natural language words that have a meaning to that application, and in a sense are similar to keywords (They form the glossary of the system.). In Figure 8-1 all the requirements are presented in this way.

The upward composition is accomplished by a two stage process:

a. Objects (abstract data types) that are acted upon by the same function are aggregated (E.g., "write personal-data" and "write business-data are aggregated" into "write personal-data, business-data".). Then, new higher level abstract data types are formed out of the aggregated abstract data types (E.g., "write personal-data, business-data" can be replaced by "write account-data".).

b. Functions that operate on the same objects are replaced by other, more general functions (E.g., "locate account", "open account", and "check account" are replaced by "fetch account".).

An automated system that performs this compositions and assists in other tasks of the methodology was developed and used. This system is discussed in Chapter 9.

Criteria for requirements composition may follow a scale that is similar to Myers's [Mye78]. Mostly, requirements should have functional and informational strength and decomposed in a way that retains the functional strength. Therefore, selection of new function and new abstract data types should be done cautiously.

To summarize, the overall method itself is independent of the way in which the requirements are presented. On the other hand, the degree of the achievable automation is strongly related to the degree of formality of the requirements. Generally speaking, the more formal the statement of the requirements, the better are the automation possibilities. The proposed method differs from the existing known method of functional hierarchical decomposition. In the suggested method, abstract users rather than requirements are being decomposed, while requirements are being upwardly composed. The benefits of the suggested method lie in the fact that the users are a more natural and obvious entity to deal with. In addition, because of its focus on users, the suggested method reduces the chances of overlooking some users (as is often done).

8.5 Checking the Requirements

Following the abstract user decomposition process and the generation of requirements for elementary users, several essential questions are in order:

a. Is the set of requirements complete? That is, have all the elementary users been identified? Has every elementary user generated all of his, her, or its requirements?

b. Are the requirements consistent? Do all the requirements conform with the users perception of the system?

These questions lead to the notion of completeness and consistency of the requirements in relation to the outer world, the user community. A complete and consistent set of requirements should refer back to the users and should be checked versus their needs and perceptions of the system.

To answer these questions, we utilize the trees that have been generated upon completion of the previous process. These trees were generated semi-independently. The abstract users tree was constructed by decomposition of the root abstract user. The abstract requirements and abstract data types trees were generated by the composition of elementary requirements, that had been generated directly by the users. Therefore, a good opportunity exists here to check the triad, the abstract users, the abstract requirements, and the abstract data types. The checks are made versus the user community in the real world. This is very important, since the other methods provide means to check the requirements only with respect to themselves, e.g., [Rob76, Kem80]. After mapping a triad, the specifier has to determine if it is logically sound. Do the abstract requirements and the abstract data type fit the specific abstract user? Are there any redundant or incorrect statements among the requirements? Is anything missing? At this point, a thorough knowledge of the system and the organization is required. At the higher levels of abstraction, the checker can be aided by a written document or operational books of the organization for which the system will be implemented.

The term "logically sound" in this context is difficult to define even though the act of checking is by itself quite simple and straightforward. If the specifier has constructed meaningful abstract requirements and abstract data types, then he should be able to see at a glance if the triad is right. Whenever a deficiency is detected, after ruling out improper upwards composition, then new requirements, new abstract users, or corrections to the existing ones can result. It is not uncommon that discovery of a deficiency can lead to new areas of investigation.

These checks can be performed during the process of constructing the abstract user and abstract data type trees. Whenever an inconsistency or incompleteness is detected, corrective action is taken and the process continues. The faults sometimes provide the specifier with a valuable source of information about spots in which further investigation is required. The banking example, when only partially performed, indicates that profitability has to be computed for every loan. While examining the abstract user "customers", the specifier can reveal that profitability could be computed timely for each account. Therefore, he may consider including this statement as a new requirement even though the idea of checking account profitability had not been raised elsewhere before.

Thus, this method indicates checks for completeness and consistency with regard to the outside world, the users community. A complete and consistent set of requirements among themselves do not necessarily ensure a realistically correct system.

8.6 Automation

Activities such as requirements composition, and checking of abstract users versus abstract requirements involve human reasoning and human expertise. Therefore, those activities cannot be fully automated. On the other hand, it is obvious that for a large software system, large data bases are formed due to the large numbers of users and requirements entailed in the system. Thus, it is suggested that the recording and handling of all those data items be automated. An important by-product is a collection of management tools that are incorporated into this automated systems.

The main automation possibilities are in:

a. construction and maintenance of the requirement tree and the abstract data type tree,

b. assistance in the composition process,

c. management aids.

Looking into Chapter 5, Section 5.5 reveals that the automation possibilities there are similar to what is mentioned here. This should not come as a surprise because these two sections propose automation of similar processes, decomposition and composition.

The requirements tree and the abstract data type tree of the specified system should be maintained together with the user tree. The process of decomposition can be assisted by a computerized expert system. It can augment the expertise of a specifier by suggesting composition patterns for requirements. Information about these patterns is collected into a knowledge base, and special algorithms called composition rules are used to process the knowledge base. An automated system named CAS (Computer Assisted Specifications) has been prototyped and tested. It is discussed in chapter 9.

The management tools keep track of the data items such as users, requirements, data types, etc. They add more information to these data items such as names, addresses, phone, location, etc. and they process tasks that deal with the interaction between the specifying team and the specified system. Some of these tasks are planning, scheduling, follow-up, etc.

8.7 Conclusions

In this chapter, the notion of a requirement has been expanded to the notion of abstract requirement in a similar manner to what was done with a user and an abstract user in Chapter 5. The specified system can now be represented in several ways by different levels of abstraction, and those different representations of the same system lead to a way to verify the system requirements.

A method that is composed of three parts is suggested in chapters 5 through 7, user decomposition, elementary requirements generation, and requirements composition. The structure of the user tree is formed by the decomposition process, and the same structure is used for requirements composition. On the other hand the elementary requirements are generated by field work that is independent of the tree structure. Therefore, this verification is done versus the real needs of the users. This kind of verification cannot be done by formal methods, because these methods do not relate to the users.

The user-oriented approach is preserved in this chapter. Requirements are composed by users and the structure of the requirements tree is the same as the structure of the user tree. The user-oriented approach seems to be logically simpler than the functional approach. A user is a better-defined and a better understood entity than a function. Beginning with users and turning to functions afterwards is more natural to the specifier and yields a better coverage of the system.

CHAPTER 9
CAS – Computer Assisted Specification

9.1 Introduction

CAS (Computer Assisted Specification) is a prototype of some tools developed in order to try out some of the suggested methodology.

The main objectives of CAS are

a. to demonstrate that the specification design method can be computer-assisted,

b. to enable experimental research utilizing CAS on real life systems,

c. to develop a better understanding of the computer-assisted requirements production method, and

d. to gain a hands-on experience with a simple expert system.

CAS enables recording and decomposition of users, recording and composition of requirements, and general manipulation of its data base. It uses a simple expert system that is based on a natural language presentation of requirements. All CAS sessions are conducted interactively with the specifier. Furthermore, the specifier is informed and consulted throughout the process.

This chapter starts with CAS tutorial that enables the reader to acquaint himself with the general notion of CAS. Then, the general concept and the design principles are discussed, and the algorithms that enable CAS to operate as an expert system are presented. A discussion of the banking example follows. This example as ran by CAS is presented in appendix B. Appendix C presents the users manual of CAS. The reader who is interested in gaining a full insight of the requirements production methodology first is advised to skip this chapter and to proceed to the next chapters, and to return to the this chapter after reading the entire thesis.

9.2 Tutorial

A brief tutorial here explains some of the basic features of the system. The users domain is a banking system that is described in figure 9-1. This example is, of course, very simplified.

The command *spc* is used in order to start a specification design or to initiate a design session:

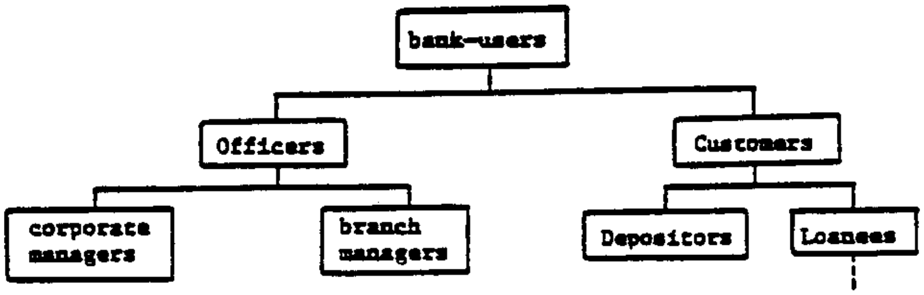

Figure 9-1: An Example--A Banking System

(1) spc bank 'bank users community'

The name of a design, which is the name of the root user, and a brief description are entered. Other users can be added to a parent using the *adu* (add user) command:

(2) adu bank customers 'bank customers'

(3) adu bank officers 'bank officers'

The user *bank* is thus decomposed to two sub-users: *customers* and *officers*. Now the user *customers* is decomposed in the same way.

(4) adu customers loanees 'persons who get a loan'

(5) adu customers depositors 'persons who deposit money'

Now, the whole users tree has been constructed.

Other commands allow maintenance of the users tree. *Dlu* (delete user) deletes a user, and *chu* (change user) changes the information or the parent of a user. If the specificer has mistakenly added the user *corporate managers* to the parent *loanees* by

(6) adu loanees 'corporate managers' 'the fat cats'

the mistake can be corrected by making the following change:

(7) chu 'corporate managers' " " officers

This command moves the user *corporate managers* to a new parent officers without changing the name and the description (two adjacent apostrophes stand for a field that is not to be changed.).

Some more facts to be noticed thus far are:

a. The specification procedure is interactive. The system responds by stating the action taken.

b. The structure of the CAS commands is similar to the structure of UNIX commands. A UNIX prompt initiates each command.

c. A command parameter is either a word or group of words enclosed between two quotes.

d. To avoid repetition of a user's name, a dot (".") can stand for the user's name as the first parameter. In this case, the user's name from the previous command is used.

e. Users are added to parent users only. Therefore, the CAS system can never have detached users. The root user is introduced by the *spc* command.

f. The command *spc* starts a specifications design or initiates a specification session of a design. Another command, *ends*, is used to terminate a design session. A session can be resumed by invoking the *spc* command with the name of an existing design. Therefore, several specification designs can reside on a system, but only one is active at any one session.

Proceeding from here, the specificer attaches requirements to the elementary users as following:

(8) adr depositors d1 write account-data

(9) adr . d2 compute balance 'compute balance after deposit'

(10) adr . d3 fetch account-data

Note that here the dot (".") stands for *depositors*.

(11) adr loanees l1 fetch loan-data

(12) adr . l2 compute payments 'payments schedule'

(13) adr . l3 compute profitability

(14) adr . l4 issue loan 'invoke the AP system'

Here the dot stands for *loanees*.

The command *adr* (add requirement) attaches a requirement to a user. A requirement is presented as an imperative sentence that has a verb and a direct object (See above.). A requirement can also have a free text that explains it further. Requirements can be manipulated by some other commands such as *dlr* (delete requirement), *chr* (change requirement), *mvr* (move requirement), and *cpr* (copy requirement). The name of a requirement must be unique within a design.

In order to compose the requirements of *depositors* and *loanees* by verb to obtain *customers*, use the following command:

(15) **cmr customers verb**

The composition by verb after the first pass yields the interim requirements
fetch account-data, loan-data
compute balance, payments, profitability
issue loan,
which are not presented to the specifier. Assuming that the knowledge base contains the following lines:
customer-data account-data loan-data
transaction-indicators balance payment profitability
The composition program goes ahead and generates the requirements
fetch customer-data
compute transaction-indicators
issue loan

All these lines are presented to you. You can decide whether to accept the recommendations of CAS or to enter your own composition. For example, you can enter the following line instead of the first line of the requirements:
fetch transaction-data
Now, if a composition by object is desired, do:

(16) **cmr customers object**

When you wish now to examine the user *customers*, you can do so by applying the command:

(17) **usr customers**
All the information pertinent to the user *customers* is displayed, its user's name and description, parent and children, and all the requirements that are attached to it (elementary requirements or composed requirements).

When you need assistance use the command:

(18) **hlp**
This commands displays a list of all CAS commands and a short explanation of their usage. In order to get a brief description about usage of any particular command, the command name should be entered, e.g.:

(19) **adr**
You will get the response:
usage: adr user-name req-name verb object text
For more detailed explanation about *adr* use the usual *man* command of UNIX:

(19) **man adr**

CAS is applied by using CAS commands interactively. The commands are detailed in the forthcoming manual pages. They are prompted by a regular UNIX prompt. A parameter to CAS commands should be a single word or group of words enclosed in quotes. The first parameter is always a user name.

The knowledge base should be installed in the file CAS0kb prior to any use of a *cmr* (compose requirements) command. It is a text file in which each line contains several words. The first word replaces the other words in the line when a match occurs. The knowledge base file can be maintained and updated by any editor.

Each command program checks the validity of its parameters and other conditions. When an error occurs, an error message is displayed and a new prompt is offered. Upon a successful completion of a command, a message that states the action that has been taken is displayed and a new prompt is offered.

Several specification designs can be stored simultaneously. Commands pertinent to a certain design should be invoked between *spc* (specification) and *ends* (end specifications). The same design can be invoked in different sessions. Each session starts by the command *spc* with the design name as the first parameter. Furthermore, a session can refer to a sub-design by calling *spc* with a parameter that is a user name in that design. In that case modifications are possible only to users that belong to the sub-tree with the given design as its root node. All design names are stored on a file CAS000 that can be updated by an editor.

CAS does not have means to update the knowledge base automatically. The specifier has to keep track of words, terms or idioms that should be added to it, and to update the knowledge base. The command *cmr* displays composition alternatives for full matches and for partial matches too. Sometimes, partial matches can indicate new desirable entries to the knowledge base.

The command *cmr* is highly interactive and consults the specifier through its execution. It enables both modes of composition, by mode and by object, and continuation of a prior composition.

9.3 CAS – A General Concept 1

CAS (Computer Assisted Specification) assists the specifier interactively in the following tasks:

a. construction and refinement of the abstract user tree,

b. assignment of requirements to abstract users,

c. composition of requirements by use of a knowledge base, and

d. general maintenance of the data base.

The design goals of CAS are:

a. to create an interactive specification environment,

b. to implement a knowledge base system,

c. to allow exercising of real-life project,

d. to develop CAS in a short time by using available tools, and

e. to develop a system that can be easily expanded and easily changed.

After evaluating several alternatives for the design of CAS the following design decisions were made:

a. The system is developed under the UNIX [UNI83] operating system and makes use of UNIX tools.

b. The CAS data base is implemented as a part of the UNIX file system. The hierarchical UNIX file system can be used to build the desired hierarchical structure of the data base.

c. The CAS system entails a set of interactive commands that are invoked by the specifier.

d. CAS commands are programmed as a set of UNIX shell scripts, using UNIX commands or UNIX programs. C is used wherever shell programming is impossible.

e. The main files: the users file, the requirements, and the abstract data type file are integrated into one file system.

f. All the information and all the data items are textual. The knowledge base is a textual UNIX file.

g. CAS is open ended, easy to enhance or to modify. It has more options than needed by the requirements production method.

h. While interacting with CAS, the specifier is informed or consulted on all the actions taken.

9.4 Design Considerations of CAS

9.4.1 General

The design requirements of CAS calls for a rapid implementation, an interactive processing and development environment that enables changes and enhancements to the system. Therefore CAS has been developed under the UNIX operating system and takes advantage of some of the features of UNIX.

9.4.2 The Data Base

The data base of CAS consists of the user and requirements trees. They are implemented as a part of the UNIX file system. A user is implemented by a directory which has the user's name and which contains a file named *user-name*.usr. This file contains user's name and description. If a user has children, then each of them has its own directory (which is a sub-directory of the parent's directory).

Each elementary requirement forms a file in the user's directory. The name of the file is *req-name*.req. It contains the verb, object and text of a requirement. Composed requirements appear in the file r002 in the relevant user's directory. The first line in this file is the mode of the composition (either 00000verb or 00000object).

Figure 9-2 exemplifies the structure of CAS data base. *Usr1* has three sons, *usr2*, *usr3* and *usr4*. *Usr2* has two sons, *usr5* and *usr6*. *Usr5* and *usr6* has requirements that appear in the files *u1.req*, *u2.req* and *r1.req*, *r2.req*, *r3.req* respectively. These requirements are composed to *usr2* and appear in the file *r002* in *usr2*'s directory. A further composition yields the file *r002* in *usr1*'s directory.

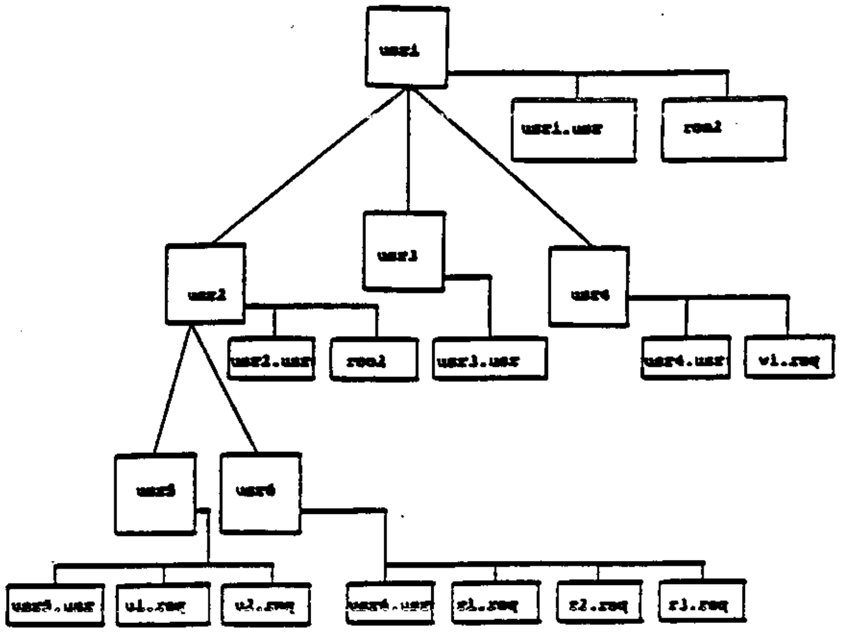

Figure 9-2: An Example of CAS File System

9.4.3 The Knowledge Base

The knowledge base is a file in the home directory and its name is CASokb. It is a simple UNIX file where each line is an entry. The first word in each line is a possible replacement for all the other words in that line. There is no distinction among verbs and objects in the knowledge base. It is not updated by CAS programs but can be created and updated by any editor.

9.4.4 CAS Files

CAS uses some other files that reside in the home directory:

CAS000, the list of all the names (root users) of the specification's designs,

CAS001, the name of the current specification design, and

CAShlp, the list of all CAS commands and their usage; it is used by the *hlp* command.

Temporary files that are used during the composition process may appear in user's directories.

r000 contains the requirements that are collected from all the children,

r001 contains the aggregated requirements (by verb or by object), as indicated by the first line of the file, and

r002 contains the composed requirements.

9.4.5 Programming

Most of the CAS system is programmed in the UNIX Shell programming language and appears as shell scripts. Running a CAS program is actually executing a script file that has UNIX commands. The part of the program *cnr* that deals with the matching of requirements with the knowledge base is programmed in C and uses the standard i/o package.

9.4.6 Installation and Maintenance

In order to install CAS the following steps have to be taken:

a. CAS commands (files) have to be installed with execution permit.

b. *CASokb* (the knowledge base) has to be installed in the home directory.

c. *CAShlp* (the help file) has to be installed in the home directory.

Maintenance of *CASokb* and *CAShlp* is done by any editor. The file *CAS000* contains all the specification designs. To remove a design, an editor has to be used.

9.5 CAS As an Expert System

One of the design objectives of CAS is to implement and to test a simple expert system. This expert system is a part of the composition process. Its role is to find abstraction alternatives using a knowledge base, and to let the specifier choose the one that seems right to her or him.

A requirement is presented as an imperative sentence that has a verb and a direct object (e.g., "check balance", "update file", "compute direction", etc.). No other restrictions exist for requirements, and it can have any words for its verbs and for its objects.

The algorithm can be summarized simply as follows: Requirements that have a common verb are grouped together and each list of objects is searched for in the knowledge base. If a match is found, then the list is replaced by the corresponding word from the knowledge base. The composition alternatives is presented to the specifier, who can choose one of them or enter one of his own composition. This process is repeated in a similar manner for a composition by object.

The expertise of the system resides in the knowledge base. It contains composition patterns from similar applications or from a general knowledge. In the present prototype, the knowledge base can be augmented manually as more experience is gathered.

A user k has the following requirements:

$$\begin{array}{ll} v_1^{(k)} & o_1^{(k)} \\ v_2^{(k)} & o_2^{(k)} \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ v_n^{(k)} & o_n^{(k)} \end{array}$$

Several users are attached to one parent, and each user has requirements. The purpose of the composition process is to deduct the parent's requirements out of the children's requirement. This process evidently yields a smaller number of more abstract requirements.

Figure 9-3 illustrates the situation before the composition is initiated. The parent $\bar{U}$ has children $U_1, U_2, \cdots U_M$. Each child has requirements that are generated either directly or by a former composition. The i-th requirement of the j-th user is

$$v_i^{(j)} \quad o_i^{(j)}$$

where $v_i^{(j)}$ is a verb and $o_i^{(j)}$ is a direct object.

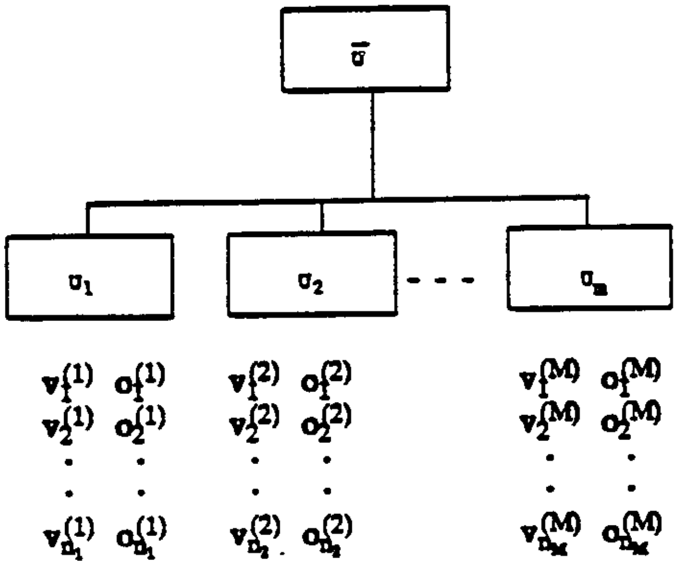

Figure 9-3: Users and Requirements

a. All the requirements of all the children are stacked (total $N = \sum_{i=1}^{M} n_i$ requirements).

b. The stack is sorted by verb. Thus requirements that have the same verb follow in the sequence. No memory of the source user is preserved.
The stack is:

$$\begin{array}{cc} v_1 & o_1 \\ v_2 & o_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ v_N & o_N \end{array} \qquad \text{(b1)}$$

(The upper indices are omitted because they indicate a link to an abstract user that no longer exists.)

c. Groups of requirements that have a common verb are unified following the verb.

$$\begin{array}{ll} v_1 & o_1 \\ v_2 & o_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ v_N & o_N \end{array} \rightarrow \begin{array}{ll} v_1 & o_1 \\ v_2 & o_2 \\ \cdot & \cdot \\ v_{i_1} & o_{i_1} \\ v_{i_1} & o_{i_1}+1 \\ \cdot & \cdot \\ v_N & o_N \end{array} \rightarrow \begin{array}{ll} v_1 & o_1 \\ v_2 & o_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ v_{i_1} & X \\ \cdot & \cdot \\ v_N & o_N \end{array} \qquad (c1)$$

where X is the vector $o_{i_1}\ o_{i_1}+1 \cdots o_{i_1}+M_1-1$.

Here, a group of M_1 requirements has a common verb v_{i_1} and therefore are unified to

$$v_{i_1} \quad o_{i_1} \quad o_{i_1}+1 \ \cdots \ o_{i_1}+M_1-1 \qquad (c2)$$

Several similar groups can occur in this assembly so its new general structure is

$$\begin{array}{lllll} v_{i_1} & o_{i_1} & o_{i_1}+1 & \cdots & o_{i_1}+M_1-1 \\ v_{i_2} & o_{i_2} & o_{i_2}+1 & \cdots & o_{i_2}+M_2-1 \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ v_{i_L} & o_{i_L} & o_{i_L}+1 & \cdots & o_{i_L}+M_L-1 \end{array} \qquad (c3)$$

Assuming that L groups are formed. The total number of objects is preserved, therefore

$$\sum_{i=1}^{L} M_i = N \qquad (c4)$$

(A group can contain a single object. Then this presentation is equivalent to the former presentation (b1).)

d. The knowledge base has the general structure

$$\begin{array}{lllll} W_1 & W_{11} & W_{12} & \cdots & W_{1K_1} \\ W_2 & W_{21} & W_{22} & \cdots & W_{2K_2} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ W_P & W_{P1} & W_{P2} & \cdots & W_{PK_P} \end{array} \qquad (d1)$$

The first word in each line W_i is the abstraction (composition) of the other words $W_{i1}, W_{i2} \cdots W_{iK_i}$. There is no distinction between an object and a verb in the knowledge base.

e. Each line in the unified requirements (c3) is compared to every line of the knowledge base. The comparison is done word by word (excluding the first word in both lines, of courses).
If the line

$$v_{i_j}\ \ o_{i_j}\ \ o_{i_{j+1}}\ \cdots\ o_{i_{j+M_j-1}} \tag{e1}$$

matches the line

$$W_L\ \ W_{L1}\ \ W_{L2}\ \cdots\ W_{LK_L} \tag{e2}$$

then (e1) is replaced by

$$v_{i_j}\ \ W_L \tag{e3}$$

However, the knowledge base line can fully match only a subset of (e1). Then, only this subset is replaced, resulting in

$$v_{i_j}\ \ W_L\ \ o_{i_x j}\ \cdots\ o_{i_y j} \tag{e4}$$

where $o_{i_x} \cdots o_{i_y}$ are the residual objects that do not match.

f. When partial match occurs, i.e., when not all elements of the knowledge base (e2) have a match in the unified requirements line (e1), then the replacement is done as in (e). The number of matches is kept, and the specifier's consent is requested.

The new assembly of requirements now has the following form

$$\begin{array}{cccccc} v_1 & o_{11} & o_{12} & \cdot\cdot & o_{1p_1} & W_{q_1} \\ v_2 & o_{21} & o_{22} & \cdot\cdot & o_{2p_2} & W_{q_2} \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot\cdot & \cdot & \cdot \\ v_L & o_{L1} & o_{L2} & \cdot\cdot & o_{Lp_L} & W_{q_L} \end{array} \tag{f1}$$

Some of the v's can be equal. In each line o or W may not appear.

g. The specifier reviews the results. He chooses one line among several lines of equal v. He may replace any combination with his own.

h. The reviewed assembly (f1) is restructured as simple sentences as in (b1)

$$\begin{array}{cc} v_1 & o_{11} \\ v_1 & o_{12} \\ \cdot & \cdot \\ \cdot & \cdot \\ v_1 & o_{1p_1} \\ v_1 & W_{1q_1} \\ v_2 & o_{21} \\ \cdot & \cdot \\ \cdot & \cdot \end{array} \tag{g1}$$

and is inverted to obtain

$$\begin{array}{ll} o_{11} & v_1 \\ o_{12} & v_1 \\ \cdot & \cdot \\ \cdot & \cdot \\ o_{1p_1} & v_1 \\ W_{1q_1} & v_1 \\ o_{21} & v_2 \\ \cdot & \cdot \\ \cdot & \cdot \end{array} \qquad (g2)$$

i. Now composition by object is done starting from the assembly (g2) and repeating steps a to f.

The composition method should be carried out starting from the lower level abstract users upwards. It is used in the verification stage of the requirements production method. This method requires that elementary requirements be attached to elementary abstract users only, and be composed upwards. However, CAS allows elementary requirements to be attached to any abstract user at any level, and blends them with the previously composed requirements before the next composition is done.

9.6 An Example: A Banking System

A banking system is used as the example by which the method is demonstrated. This system falls within the definition of a large software system because it has a large and diversified community of users. The example is simplified and incomplete in comparison with a real application. It serves only as an illustration to the proposed method.

Figure 5-2 in Chapter 5 shows the results of the decomposition procedure, which is the user domain of the banking system (steps 1 to 3). The root level, level 0, is the most general abstract user called "User Community". It is decomposed into four abstract users: "Branch", "Customers", "Adjacent-Systems", and "Bank-Management". Decomposition proceeds to level 2 for all level 1 abstract users, and further to level 3 for "Depositors" and "Loanees".

The decomposition is not carried out to the same level on every abstract user. It is stopped whenever it seems to be logically right, i.e. where further action is impossible or not necessary. A good indication for stopping the decomposition process is arriving at an abstract user that can be directly approached and interrogated for requirements.

In this example, it is quite difficult to approach a generalized customer of the bank or even a loanee. However, in order to understand the requirements of a mortgage loanee one can approach the bank officer that is in charge of mortgages and interview him. Should that be too generalized in the sense that no single officer handles this task, the abstract user "loanee mortgage" can be further decomposed into "first home loanee", "home improvement loanee", "home dealer loanee", and so on. This decomposition can point to the bank officers that can be interviewed. This process replaces the analogous process of decomposing the abstract requirement "provide a mortgage" into a collection of simpler requirements.

At lower levels, an abstract user and an abstract requirement are sometimes interchangeable. The abstract user "Depositors" has four sub-users "Dep-New-Account", "Dep-Old-Accounts", "Dep-Other-Branch" and "Dep-Other-Bank". However, the requirements can be interpreted as requirements stemming from the abstract user Depositors. In order to ensure coverage and simplicity it is advisable though to perform the deepest decomposition possible. It is, after all, easier to eliminate unnecessary functions later than to add necessary functions later.

Examination of the results reveals that the requirement "Check Profitability" that is applied to loans can also be applied to accounts. Since it is a logically desirable requirement, it should be implemented in the system. This requirement could have emerged otherwise from the abstract user "Control". The control department of the bank might has imposed requirements upon the system that force any transaction such as loans, and any financial entity such as account, branch, etc. be analyzed for profitability. Secondly, the requirement must be carried out by another system called "Accounts-Payable". Thus, a new abstract user has been detected by the method. Should this abstract user be forgotten in the initial layout of the specification, then it would have emerged elsewhere as a result of the abstract user "Adjacent Systems". So the proposed method has demonstrated its ability to guide the checking of the requirements for completeness.

As mentioned, Appendix B presents a full session of CAS running the banking system example. Two different composition presentations are shown, one presentation is by verb and object and the other by natural language.

9.7 Conclusions

The following conclusions have been arrived at by using CAS for several applications.

a. CAS has demonstrated that at least part of the methodology of requirements production can be automated, thus forming a computer-assisted system.

b. CAS has many advantages in performing the clerical and administrative tasks of the specification phase.

c. Representation of requirements and specifications in natural language makes CAS accessible to all the participants of the process.

d. The use of a relatively simple expert system as a part of CAS brings a member of advantages to the specifiers.

e. The design goals of CAS namely those of interactive operation, integrated system for process and management, etc..

CHAPTER 10
Stage 6: Functional Specification

10.1 Introduction

The previous stage, raw requirements processing, deals with the conversion of a bulk of investigation summaries into requirements and the classification of these requirements into main subjects. In addition, the requirements are rated by the users to designate their relative importance.

The purpose of the present stage, the functional specification stage, is to consolidate a set of hierarchical functional specifications out of the new requirements. In other words, in this stage a transition is made from user-oriented requirements to function-oriented requirements. The specified system can now be perceived from two different views, the users' views and the functions' views.

The product of this stage is a specifications document. It is described in appendix A.

10.2 Method

The functional specifications stage involves

a. determination of main functions,

b. mapping of users and functions into requirements,

c. analysis of functions,

d. application of special techniques such as scenario analysis or modeling if necessary, and

e. preparation of a specifications document.

As the former stages of the process, no assumption is made on the way the requirements are stated. But evidently a natural language is the most common medium for stating requirements. Other presentations are not precluded and can be applied towards the end of this stage.

This stage involves mainly human reasoning and analysis, and is one of the most creative stages of the entire requirements analysis method. Very little here can be automated, but extensive use can be made of the stored results of the previous stages.

10.3 Classification of Requirements to Functions

All the requirements should be classified into a small number of functions or subjects. A raw classification was done in the previous stage and is discussed in Chapter 7. The specifier has now to determine the main functions of the desired system. Some criteria have to be applied:

a. The total number of functions should be manageable and therefore should be fewer than ten.

b. One of the subject /functions should be called system-requirements to represent system level requirements that do not belong to any specific subject, e.g., security, maintainability, etc..

c. The subjects should represent functional, rather than technical classification. E.g., for a logistic system, good functional classifications would be those of inventory, purchasing, and maintenance, rather than those of update, retrieval, reporting, etc.

d. There should be a full coverage of the system by the subjects /functions. No raw requirement is to remain unclassified.

e. Care must be taken when the specified system is organization-oriented. Generally, classification by organizational lines is not desirable unless it represents a functional classification (e.g., the purchasing department may coincide with the purchasing function).

The overall classification process is illustrated in Figure 10-1.

10.4 Functional Analysis

The purpose of this stage is to analyze the requirements that were formerly assigned to a subject, to identify the interactions and the commonalities among them, and to form a hierarchical functional framework of the specified system. In other words, the purpose of this task is to convert a set of user-oriented requirements into functional specifications.

This task is unstructured and is mainly done by human reasoning. The specifying team is organized into work teams, one for each subject. Whenever two subjects are assigned to the same team, they should be functionally related. An assembly of all the work groups should take place regularly to discuss work progress, inter-group subjects, and general system specifications.

Guidelines for the methodology for the functional analysis are as follows:

a. The analysis is a top-down functional analysis done by hierarchical decomposition, as described by several authors ([Alf77], [Bel76], [Bel77], [Dav79], [Ham76]).

b. The specifications are presented as a subject tree.

c. During the analysis, requirements can be found to be inconsistent (one requirement contradicts another requirement) or incomplete. In this case, the corresponding user or users should be identified and be referred back to.

d. During the analysis, the work team may identify requirements that belong to another subject or affect another subject. Then, transfer of requirements or definition of interfaces is desired.

e. A program design language [Cai75] is a powerful and an easy-to-use tool to support this analysis.

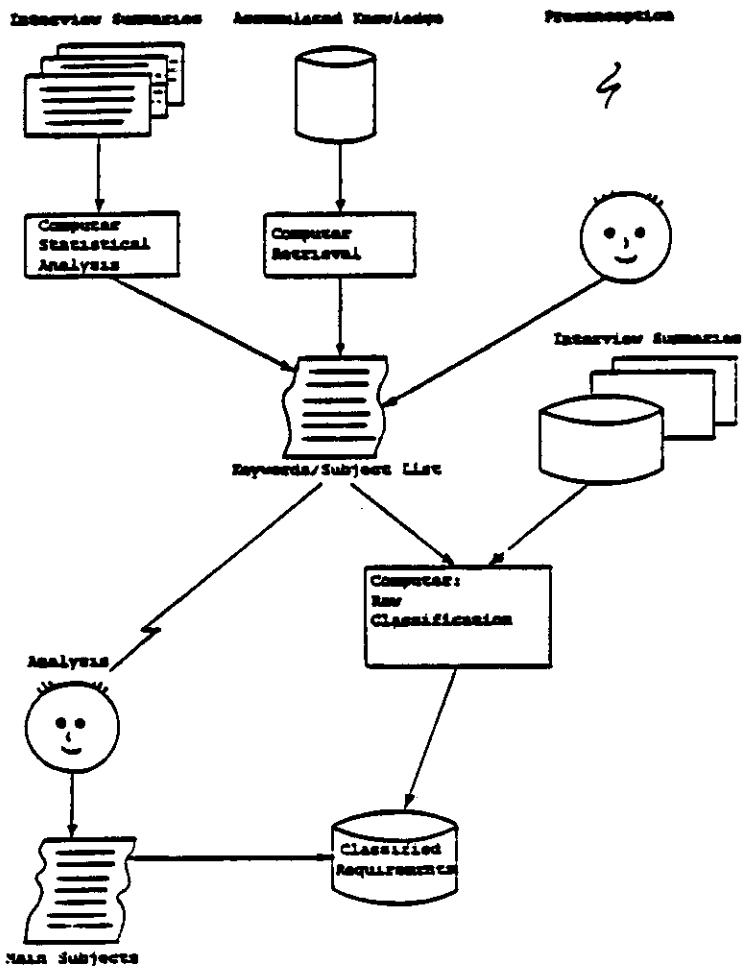

Figure 10-1: Classification of Requirements to Functions

It has been shown to be a superior tool in compare to flowcharting in the detailed design phase [Ram83], and this applies to requirements too. It allows a semi-formal representation, with embedded descriptions in natural language. A recently developed program design language [Bur83] allows the task of hierarchical decomposition to be performed and checked interactively.

f. At this point, formal specifications can be introduced. Various reasons may lead to a decision to fully formalize the specifications. Among them are the need for precision and the need for internal verification. It is suggested though that a natural language presentation be maintained together with the formal representation. Some authors show how to start from informal requirements and to formal requirements [Bei84] or how to start with informal but precise English description and get a representation of the data types, variables, operators, etc. from the structure of the English sentence [Abb83].

g. The depth of the decomposition is determined by the depth of the user decomposition and by the scope of the specified system. Care must be exercised in order not to get into design analysis or design details.

h. Special techniques may be used for various parts of the specified system, such as analysis-by-scenario, modeling, rapid prototyping, etc..

10.5 System Representation

The method that is discussed in this chapter defines the specified system as a set of functional specifications. In a previous chapter (Chapter 7), the system is defined as a set of user-oriented requirements. Both definitions of the specified system are identical if the analysis is done properly. Thus, two different representations of the same system are generated as illustrated in figure 10-2.

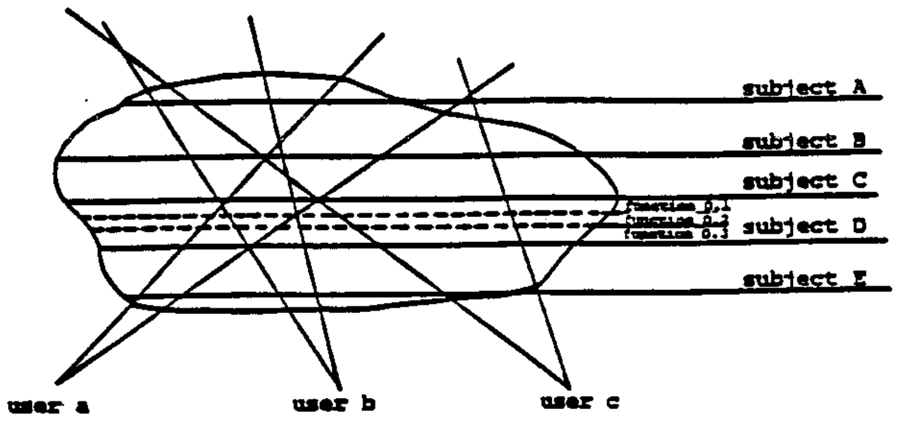

Figure 10-2: System Representation by Users Views and by Subjects

In this figure,the system is represented by (or divided to) subjects *A, B, C,* and *D*. Subject *D* is shown to be subdivided into functions *D.1, D.2, D.3*. On the other hand, the system is covered by users views *a, b, c*. These user views can cross the subject /functional borderlines. User views do not have to cover the system entirely. It is possible that the specifying team adds, for various reasons, requirements that are not uttered by any user, e.g., due to anticipation of future enhancements.

Each representation has its advantages for different uses. The user-oriented representation is best understood by the users and is easy to maintain for a system that is rapidly changing. When new users or groups of users are frequently added to a system, in other words, when the system grows significantly as time passes, then the users-oriented representation is better. However, from the system designers' and programmers' points of view, the functional representation is more natural and easier to work with.

Systems do change and grow as time passes. New users, new requirements and changes to the specifications are introduced. Although it is difficult to keep and maintain both representations and to keep them consistent with each other, it is highly recommended to do so. The tremendous effort that is invested in system maintenance justifies this redundancy. There are real dangers in not maintaining both requirements, as it is easy to lose the connection between the users and the requirements.

10.6 Automation

The functional specification stage involves mainly human, unstructured activities and not much of its activities can be automated. Yet, for large systems, the computer supports the specifying team by helping to keep track of all the requirements and thus ensuring that no requirement is left behind.

The method of functional analysis that is done by hierarchical decomposition can be supported by a tool called a program design language [Cai75]. The main idea is that a small set of primitive constructs such as **if then else**, **do while**, etc., together with means to describe abstract data types and text written in a natural language can be used to describe a program, and therefore be used as a design tool. This idea was further developed to accommodate several abstract data types and to enable hierarchical decomposition and some verification [BeY82]. It was shown that an interactive program design language [Bur83] is simple to use and powerful as a design tool. The use of a program design language can be extended to describe and to support the analysis of functional specifications, because it includes all the necessary components. It is therefore recommended that a program design language be included as one of several tools to support computer assisted specifications.

10.7 Conclusions

This chapter shows how to convert raw, users-oriented requirements into functional specifications. These specifications can later be used as the input to the design phase. The process shows that there are two ways to describe a system. One way is by a set of user views. A user's view is a set of requirements that are attached to, or generated by a user. The other representation is by functional decomposition, i.e., the system is presented as a set of functions, or system properties.

The analysis work of this stage can be supported by a program design language. Furthermore, an automated system can help in keeping track of all the requirements and in performing other management type activities.

CHAPTER 11
Stage 7: Operational System Specification

11.1 Introduction

This stage produces the last product of the requirements production method, the operational system specifications. The main purpose of this stage is to represent the specified system as a component of a global environment. The system participates in processes that are common to the environment and to itself. These processes constitute a third representation of the specified system

In this chapter, the method is outlined and then detailed. A description of the final document is given in appendix A. As in every chapter, one section is dedicated to describing the possibilities for automation. This chapter ends the description of the method to produce requirements and specifications for large software systems.

11.2 Purpose

The essence of operational specifications is to identify and describe basic system functions and system processes, to show how the system interacts with its environment, and to deal with derived topics such as organizational and operational restructuring, documentation, training, security, maintainability, etc. This constitutes a higher abstraction than the functional specifications.

11.3 The Method

A higher level presentation of the designed system is desired. This is a presentation by basic operational functions and processes. The system is perceived as a part of a larger entity where global processes, that pertain to the system and its environment, use system functions in order to achieve the operational goals.

A *basic operational function* is defined to be a system component that is used by the system users. A *system process* is defined as a basic operation common to the system and its environment that is performed in order to achieve one or more of the operational goals. A system process invokes a series of basic operational functions. Figure 11-1 gives some examples to bring these definitions down-to-earth.

The basic methodology of this stage can be described as:

present state + requirements + constraints → new state

The present state is revealed from the information that is gathered in the interviewing stage (See Chapter 6.). The requirements are those of the functional specification stage (Chapter 10). The constraints can be regarded as a form of requirement and are revealed during the interviews and processed afterwards.

Specified System	Basic Operational Functions	System Processes
Banking	Calculation of interest Daily balance sheet Cash Positioning	Loan process Deposit process Inter-branch operations
Air Command and Control	Tracking Positioning Display	IFF process Dogfight Missile launching

Figure 11-1: Examples of BOF and of SP

The basic steps of this stage are:

a. classification of basic operational functions, processes and users (in the present state and in the designed function),

b. mapping of the processed requirements into basic operational functions and processes,

c. deriving the general system concept,

d. specification of every basic operational function and of every process,

e. presenting the users' views, and

f. deriving some miscellaneous subjects.

Each of these basic steps is detailed now, and a description of the final document is given later.

11.4 Classification

The first step of this stage is to classify the basic functions, the processes and the users both in the old system and in the new, designed system. The term "old system" needs a little elaboration. Whenever a system is designed to replace a present system that is in operation the term "old system" is evident and refers to the present system. When there is no automated system to be replaced by the specified system, than the extant manual system can be considered as the old system.

The classification is not necessarily the same in the present (old) and in the designed (new) system. Changes in the designed system can cause addition, deletion, unification and splitting of functions, processes, and users. The way to perform this classification is by referring back to the sources that are mentioned before, i. e., the interview summaries and the processed requirement. Most of the necessary work about functions and users has already been done previously. Users were identified by a process of hierarchical decomposition, that leads naturally to classification. Requirements were composed during the verification stage (Chapter 8), and this abstraction can indicate a classification. Furthermore, functions were classified during the functional specification stage (Chapter 9).

Basic processes have to be identified. If the knowledge about the system processes is incomplete, then reiteration of interviews is necessary. This classification can be done right after the interviews. However, the operational system specifications stage may reveal that some of the processes are unnecessary, and therefore this classification should be done as the last step of the requirements production method. Going back to the users at this point of time should not be considered bad because both the users and the specifying team can benefit from increasing the points of contact.

11.5 Mapping

This stage is similar to the mapping sub-stage of the functional specification stage (Chapter 10). Every requirement has to be mapped to a basic operational function and to a basic process. This task is done by the entire specifying team according to the following guidelines:

a. Every requirement has to be mapped to at least one basic operational function and to at least one basic process.

b. One requirement can affect more than one function /process.

c. Reclassification of functions /processes can occur during the process of mapping, and to cause a different distribution of requirements to classes.

d. Some requirements are found to be mapped to a large number of functions /processes. These requirements pertain to general system requirements.

The representation of the designed system by basic operational functions and processes is somehow different from the representation that is suggested in the previous chapter (9.6–System Representation). Here, the system is perceived as a component of a larger entity, environment, organizational structure, the whole world, etc. that participates, partly or as a whole, in processes that are invoked by that entity in order to achieve a goal. The goal is determined by the type of that entity. The processes are only partly internal to the designed system. They can pass the borderline of one system and involve the environment or other systems.

Figure 11-2 illustrates the new representation of a system, together with the other representations.

The processes are shown to go across the system, and each process occupies only a part of the system. A good specification design rationalizes one system to include full processes whenever possible.

11.6 A General System Concept

Now the whole specifying team is engaged with delineating the general system concept. At this point in the specification design process, all the information is known, and a clear framework of the designed system emerges from the processed requirements.

The general system concept has to determine

a. the purpose and operational goals of the system,

b. the criteria for checking the operational contribution of the system,

c. measures for testing the compliance of the system with its goals, and

d. the functional principles of the system.

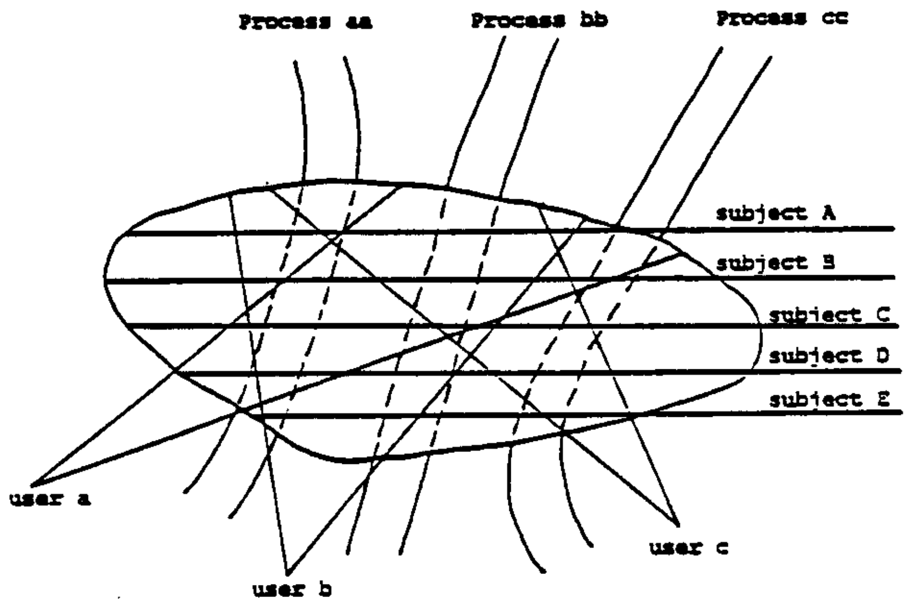

Figure 11-2: Three Alternative Representations of a System

Traditionally, system developers expect to delineate the system concept prior to commencing any work on the system. However, many times this has been proved to be impossible. The case of a large software systems is clearly one example in which only few prior directions of that sort can be given, if at all. The method that is described here reveals only after several iterations, refinements, and abstractions, how to arrive at the system concept.

This step is done by human reasoning. There is no recipe for doing it. However, it should not be a difficult task to accomplish because the previous stages have made all the information available and properly organized for the specifying team.

11.7 Specification of Basic Operational Functions and of Operational Processes

11.7.1 Methodology

The task of this step is to determine each function /process for the designed system by taking the following actions

a. Documenting the existing (old) function /process, if such a function /process exists.

b. Analyzing the functional specifications that pertain to the function /process. Those requirements have been mapped to the function /process before.

c. Determining the function /process for the specified (new) system.

d. Documenting the specified function /process.

11.7.2 Team Organization

The specifying team is divided into smaller groups. Each group is assigned one or more functions/ processes of a similar nature. The size of each group depends on the task assigned to it. Each group should include at least two persons. One of the groups deals with general system concepts, criteria and measures as mentioned in section 11.6.

11.7.3 Work Procedures

a. A working group determines and documents each function /process assigned to it. Each group is free to determine its own work procedures. However, schedules are to be imposed by the project manager. The results are documented in a uniform way to be discussed later.

b. Interaction between two functions /processes are discussed mutually by the two relevant groups, unless both functions /processes are assigned to the same group.

c. Main issues or problems are discussed by the whole specifying team.

d. The whole specifying team meets weekly to evaluate and approve the products of the work groups, and to discuss subjects that are of general concern.

11.7.4 Documentation

The way the functions /processes are documented should be uniform for a specifications design. The suggested requirements production method does not impose any particular form presentation, but encourages compliance with certain recommendations:

a. Both graphic and written description of each function /process are recommended.

b. The graphic presentation can be a HIPO chart [HIP74] or a structural chart such as the one used in Yourdon's methodology [DeM78].

c. Other presentations, such as a formal presentation, can be added if necessary.

This documentation, after being reviewed and approved, comprises a part of the final product as is discussed in appendix A.

11.8 Miscellaneous

After completing the determination of basic operational functions and the processes in the specified system, other subjects can be considered and documented in the final report. They are described here.

11.8.1 Organizational and Operational Changes

Implementation of a new system, especially a large system requires organizational and operational changes to be carried out by the system users. This step defines and determines what preparations the users have to make in order to ensure a smooth operation. The source for this task is the user tree and the mapping of processed requirements to users.

11.8.2 Documentation and Training

The representation of the system enables users to determine what kind of documentation and what kind of training every user needs. The results of this primary investigations leads later to the preparation of a documentation and a training plan. The detailed work is done at later phases of the system development.

11.8.3 Data Directory

The presentation of the requirements, that is discussed in Chapter 8, gives rise to all the abstract data types that appear in the specified system at various levels of abstraction. This is because each requirement is presented by a verb and an object, and the object turns out to be an abstract data object. Other presentations of requirements are, of course, possible, but eventually data types will emerge from any presentation. These data types are used in the descriptions of the functions /processes. A list of all the abstract data types of the system is an essential input to the design phase in the system development, as well as in the description of requirements. Good design work will derive the system's data directory and the logical structure of the system's data base out of the abstract data types.

11.9 Automation

This stage, which is the last stage of the specification design, deals mainly with the preparation of the final document. Obviously, a text processor is the most used tool for this stage. A text processor is used in several stages of the process starting from input of interview summaries.

Several information items that are gathered or generated in previous stages are used here. These items are the user tree, the requirements tree, with abstract data types, processed requirements, etc. This information is accessed by retrieval tools.

Classification and mapping of requirements appear to be a repetitive actions in several chapters. A software tool can be provided to support it. It is discussed in the automation chapter (Chapter 12).

11.10 Conclusions

This stage concludes the process of production of specifications for large software systems. It deals mainly with yet another presentation of the system and with defining the position of the system in its environment. The method here draws upon the information that is gathered and processed in the previous stages. It shares some similar processes (such as classification and mapping) with previous process.

The operational system specification stage can be computer-supported in several of its tasks such as word processing and document preparation, information retrieval, classification and mapping.

CHAPTER 12
Automation of the Requirements Production Method

12.1 Introduction

The role of an automated system during the specifications process is the role of a supportive system. Such a system accompanies the specifiers, provides them with information and tools to process the information and, whenever possible, humbly suggests alternatives or courses of action. In this last role, it acts as an expert system [Bar82].

Management and control of a specification project, that usually is a part of a larger software or system project, involve keeping track of users, requirements, data types, specifications, etc., mostly unstructured, and other tasks such as scheduling, document handling and dissemination, generation of work plans, and ensuring a smooth transition to the design phase.

It has been argued that significant productivity gains require an integrated program of initiatives in several areas. One of them is a software support environment. Tools for requirements production are a part of the process technology for the 1990's as discussed in [Boe83]. The requirements production method is valuable by itself, but undoubtedly its contribution is larger when it is supported by automated tools.

This chapter attempts to analyze and to present the functional characteristics of a Computer Assisted Specification System (to be referred to as CASS). The next section (12.2) explains the methodology used for obtaining the requirements of such a system. Afterwards, the general concept of the system is presented (12.3) and a more detailed description of each service/ function is given.

Several interesting topics of CASS were subject to a more thorough research. This was done by a prototyping of a sub-system named CAS (Computer Aided Specifications) and running it on a typical specifications design problem. The CAS system is described in a previous chapter (chapter 9).

12.2 Methodology

A good way to arrive at the specifications of CASS is to use the requirements production method described in the previous chapters. In order to save time and space, many shortcuts are done here and the steps taken are

a. definition of the users domain of CASS,

b. listing of automation functions of each stage,

c. formation of subjects from common functions,

d. analysis of each subject,

e. analysis of the management subject,

f. discussion of CASS's main components (data bases, knowledge base, tools), and

g. listing CASS services.

As mentioned before, this chapter attempts to present only a functional delineation of CASS. Therefore, no design of CASS is presented here.

12.3 Automation Needs by Stages

Figure 12-1 presents a list of actions that can be automated in the specification process and in the management of the process. The list is organized by stage.

Stage	Specification Automation	Management Automation
1. Study		
2. User Identification	Decomposition Maintenance of users file	
3. Interviewing	Word processing Document processing Document handling	Task assignment Scheduling
4. Raw requirements processing	Text analysis Classification Correction of problems Maintenance of req. file	Distribution of documents Handling of mailing list
5. Verification	Composition Maintenance of req. file Handling of verification algorithms	Errors processing Statistics Completeness
6. Functional Specification	Subjects classification Text processing	Document processing Version handling
7. Operational system Specification	Classification and mapping Text processing and editing	Cost modeling
8. General CASS		Work plans Team assignment Keeping track on items Guidance and help

Figure 12-1: Candidate Issues for Automation

The *Stage* and *Specification Automation* columns of this table are compiled from the description of various stages of the process. For more details, the reader is referred to the corresponding chapters. The *Management Automation* column describes management activities that are discussed later in a special section (12.11.8).

From the above table, the services that CASS should provide can be listed:

a. Text and document processing

b. Text analysis

c. Classification and mapping

d. Decomposition

e. Composition

f. Program design language processing

g. Maintenance of the information base

h. Project management

i. Process monitor

Each of these subjects is described and analyzed subsequently, after the description of the general system concept.

12.4 User Domain of CASS

The users domain of CASS is described in figure 12-1. The main users of the system are the specifiers. As abstract users, they can be further subdivided into specifiers at the decomposition stage, specifiers at the verification stage, etc. Each kind of specifier requires different services from the system, as is described previously. The system implementors can be considered to be CASS users because they are the persons that use the product of the methodology. Naturally, the potential users of the specified system are users of CASS, together with the customer and the project manager.

An elaborate analysis of the users domain will probably reveal ideas for augmenting CASS to make it usable by users, designers, etc. Here the system is focused on the specifying team and its management.

12.5 A General Concept of CASS

CASS is an interactive software system that provides on-line services for specifiers and for project managers. These services enable them to conduct an orderly methodology designed to obtain system requirements and specifications. The system can be described by three layers, information bases, tools, and services, as shown in figure 12-3.

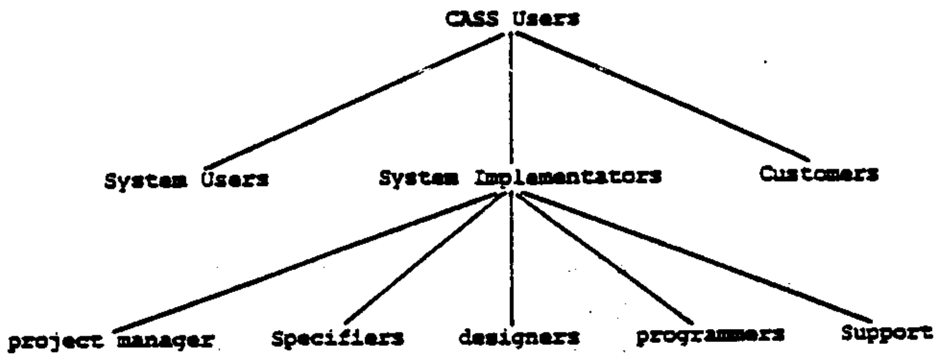

Figure 12-2: Users Domain of CASS

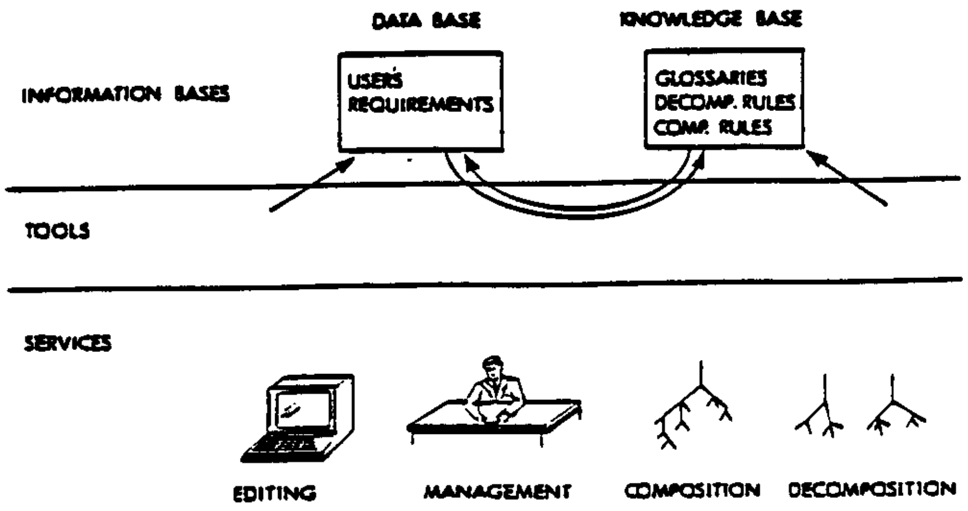

Figure 12-3: An Outline of CASS

The information bases, the data base and the knowledge base, contain information about the designed system and instructions for its processing. A tool is a program that perform a certain action or process, typically, on the information bases. A service, or application, is the result of applying one or more tools in a certain order.

12.6 The Data Base

The CASS data base contains all the information that is relevant to the specified system. It is further subdivided into files as following:

User File: Contains all abstract user records and their relationships, e.g., parent-child. This includes relevant user information, such as name, description, identification, type, references, etc., and management information, such as date and reason for specifying that user, interviewer, dates for planned interviews, etc.

Interview Summaries File: Contains all interview summaries, references, and management information.

Requirements File: Contains the requirements that are generated by the elementary users and requirements that are generated by composition. This file can contain various presentations of the same requirement and is conceptually linked to the user file and to the interview summaries file. The same file holds raw requirements and processed requirements. Management information such as dates, assignments, references, rating etc., accompanies each requirement.

Document File: Every document that is produced by the process resides on a file. Several revisions of a document can be held concurrently.

Abstract Data Type File: The data of this file is embedded in the requirements file. However, during the requirements composition process, the file is created to contain the data types that are part of the future system.

12.7 The Knowledge Base

The knowledge base of CASS contains knowledge that pertains not only to a specific application, but to a larger class of applications, to other similar applications, or to the environment of the designed system. It attempts to provide the specifiers with expertise and prior knowledge of other systems that were not designed by them.

The knowledge base is used by some of CASS services, most notably the composition, text analysis, classification and mapping. It has two main parts, the application glossary and the rules.

The application glossary is a list of words, phrases and terms that are pertinent to an application or to a class of applications. Data items can be organized in the glossary either separately or as a part of a structure, e.g., a hierarchy or a semantic network. The hierarchy and the networks are used to indicate how terms are abstracted or decomposed and how they can be meaningfully chained.

The rules indicate how the application glossary affects the data base. For example, a simple yet useful algorithm of composition is to match a list of words from the data base to the same list in the application glossary and to replace the list by the one word provided by the glossary for the list. This was actually done in the CAS prototype (See Chapter 9.). Composition rules may be more complex, and other rules may apply to text processing or mapping. For example, a simple rule to identify classes of requirements out of interview summaries would be to count the frequency of the words and phrases in the text, and to build classes out of the most frequent words and phrases (discarding words such as "the", "and", "is", etc.).

12.8 Tools

The tools are higher order elements of software that perform a specified task, usually by operating on the information bases. The tools are described together with the services.

a. The DBT (Data Base Tool) provides means to access the data base, to update, add, delete, edit, and retrieve abstract users and requirements, and to generate reports.

b. The KBT (Knowledge Base Tool) provides means to generate and access the knowledge base, to edit and update items in the application glossary and the rules, and to retrieve whatever necessary.

c. The KOD (Knowledge Base On Data Base Tool) provides means to apply the knowledge base to the data base. This tool is a part of the decomposition and composition services, and it can be applied either to users, requirements, or functions.

d. The DOK (Data Base On Knowledge Base Tool) provides means to enrich the knowledge base by augmenting it with the current or the latest specifier decisions. Actions that are taken on the data base are analyzed and more terms are added to the application glossary. Thus, CASS can be seen as a system that learns from its own experience.

12.9 Text and Document Processing

Most of the data processed by the specification method is worded in some natural language. Therefore, the CASS system should be able to accept input in that form. As explained previously, a natural language is the most common way to express requirements and specifications and is almost the only form that can be understood by all participants in the process, the specifiers, users, and programmers, etc.

Therefore, one of the CASS services is text and document processing that enables entering text, word processing and editing, document formatting, typesetting, etc. Some of the information should be distributed to specifiers, users, etc., and therefore, electronic mail could be of an advantage.

The text and document processing service is used at most of the stages of the requirements production method. A good example of such a service is offered by any UNIX operating system [UNI83].

12.10 Text Analysis

The text that is entered into the CASS system has to be processed and analyzed. A relatively simple analysis of text can result satisfactory results. The following sections describe the required analysis.

12.10.1 Identification of Keywords

The first classification of interview summaries (chapter 7) is done by identifying keywords and assigning parts of a document to these keywords.

One method to generate these keywords is by scanning the text, counting the frequency of each word and eliminating noise words. Meaningful keywords that lead to classes probably appear among the most abundant words.

A list of other keywords reside in the knowledge base, as a result of previous analyses of similar systems. A third source of keywords is the specifier's preconception.

This service can be enhanced by dealing with phrases rather than words, and by using more elaborate methods of comparison, e. g., to strip words and compare only their stems. A discussion of skimming techniques can be found in [Cul78] and in [DeJ79].

12.10.2 Classification

Classification of text is a service that is needed at several stages of the method, the raw requirements stage, the functional specifications stage, and the operational system specifications stage. The task is to assign or to map a portion of text to a phrase or a keyword.

One technique of doing the mapping is to scan the text for occurrence of the keywords. A more elaborate way is to use parsing techniques on the sentences of the text and to replace words or phrases by an equivalent phrase. No automated technique can guarantee a full classification. The least that such a service can do is to let the specifier do the classification and just to record and store the mapping.

Figure 12-4 illustrates the keyword identification and text classification services.

12.11 Mapping

Mapping means establishing a correspondence between two elements of the system, e.g., of requirements to subjects, requirements to processes, etc. This mapping can be done sometimes by text analysis or identification of matching keywords. However, usually it must be done by the specifier.

The method requires several mappings to be established and maintained, and this is a service that CASS can provide quite easily. Thus, the specifier is free from the need to do tedious, mechanical tasks.

12.12 Decomposition

Decomposition or refinement is replacing an element of CASS by several, more detailed elements. These elements are an abstract user, an abstract requirement, a processed requirement, etc.

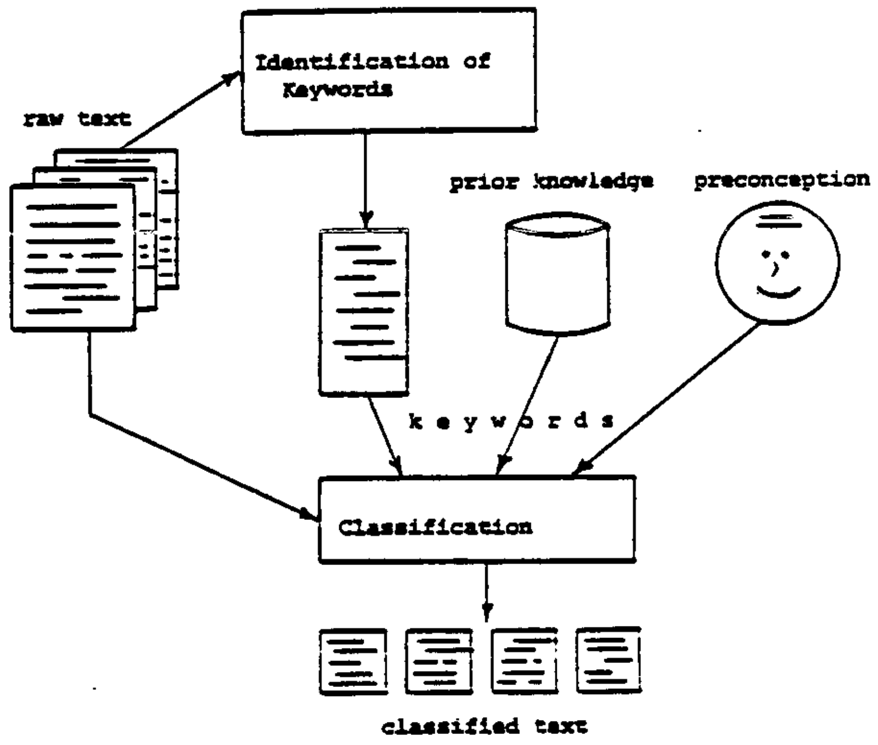

Figure 12-4: Keywords Identification and Classification Services

This process involves human reasoning, and, therefore is done by the system specifier. CASS enables interactive communication with the assembly of processed elements, monitoring, and some error detection. Prior knowledge can be stored in CASS to assist the specifier. This prior knowledge consists essentially of decomposition patterns that have been found to be frequently employed in similar applications.

The discussion in the next section about the composition service is relevant to this section, too.

12.13 Composition

Composition or abstraction is replacing an assembly of elements by fewer elements. This yields a less detailed and a more general description the whole assembly. Composition is used in the specification process during the verification stage, in which elementary requirements are composed to higher level requirements and checked.

This process, like the decomposition process, involves human reasoning and is done mainly by the specifier. CASS assists by taking care of all the clerical tasks of recording and by availing prior knowledge or expert assistance.

The knowledge base is described in section 12.4.4. The composition algorithm combines all the composed elements, analyzes them and refers to knowledge base during this analysis. Chapter 9 describes a straightforward algorithm in which the composition process is done by replacement of a string of words by a single word.

Generally, composition rules of textual elements can be referred to in three levels, the word level, the phrase level, and the structure level. Each level assumes a different level of composition, by words, by phrases or parts of sentences, or by structures that are sentences or other high order descriptions such as graphs, tables, etc.

The application glossary can be divided to application-oriented terms, computerese terms, and general, natural language terms. The first class contains terms that belong to the application itself. The second class has to do with general computer jargon that often infiltrates the requirements, and the third class consists of plain natural language terms. Therefore, only the the applications glossary has to be constructed for each class of applications (e.g., banking, logistics, air traffic, etc.), whereas the second class is built once for each type of designed system such as, real-time systems, operating systems, data-processing systems, etc., and the third class pertains to all designed system and has to be constructed only once.

To summarize, the composition and decomposition services of CASS act as an expert system. Thus, prior knowledge and expertise as recorded by CASS can augment the capabilities of the specifier in performing composition and decomposition.

12.14 Program Design Language

Program design languages have been in use for several years [Cai75]. The practical experience gained shows significant improvement in productivity and ease of use in the design, checkout, and integration phases of a software project. A program design language can be used for statement of functional requirements in a similar way to design. In CASS, a program design language processor is used as a service in the functional specification stage (Chapter 9). The functional requirements are written down and then refined and analyzed by the program design language processor.

Contemporary program design languages such as SuperPDL [Bur83] are better fit to CASS and to be used for specifications presentation and analysis. Some of their characteristics are

a. an interactive environment, so that the work is done interactively with the designer; furthermore, several designers can interact with the same design,

b. syntax directed editor, which enables on-line analysis of constructs,

c. design analyzer, which checks completeness and intermodule interfaces,

d. abstract data types, which are used to describe and manipulate data and actions, and

e. hierarchical structures, which can be maintained and refined.

The way of representing a design by a program design language is by using several primitive constructs such, as while, select, call, etc., and definition of data types. All the information that is embedded within a construct is stated in natural language. Possibly the same information can be processed by the text processing and text analysis services that are mentioned previously.

12.15 Information Base Maintenance

The information bases of CASS, the data base and the knowledge base, are updated during the specifications design process. This update is done by one of the services mentioned above. However, a service of straightforward maintenance of the information bases is needed and is frequently used.

The main features of this service are outlined:

a. Each information base and file (as mentioned in 12.4.3 and 12.4.4) is accessible to the user.

b. Each data item and knowledge item can be added, deleted or modified.

c. Each link and relationship between data items or knowledge items can be modified.

d. Whenever possible, a syntax directed editor is incorporated into this service. It enables error detection and is easy to use.

e. This service should preserve consistency in the designed system. Thus, no direct modifications that remove a prior consistency are allowed.

f. This service should enable concurrent maintenance of several revisions of one designed system.

g. Management data and files are maintained by this same service.

12.16 Project Management

Large software systems are difficult to manage mainly because of the unstructured nature of the work and the complexity of the desired system. It is important to have good management data, tools and practices from the beginning.

Managing the specification design phase of the project is a difficult task by itself. It is only partly structured, and it involves numerous data items, users, requirements, data types, processes, basic functions, etc., and many persons, i.e., specifiers, users, customers, etc. Proper management practices should accompany any methodology, to ensure proper implementation of the methodology.

It is suggested here that one system be implemented for the specifications design process and for the management process. The benefits of combining management tasks and specification tasks in one system are quite evident. Redundancy is avoided, project management has a better visibility to the whole process, no conflicts of information occur, and reporting becomes more reliable.

The meaning of project management here implies an extra dimension of the system by incorporating a part of its environment, namely the specifiers, users and the organization, and establishing the relationship between these entities and the elements of the system.

The main tasks of the management service are presented below. They are the output of Section 12.3.

a. Inventory Control: keeping track of all the data items of CASS, users, requirements, data types, processes, functions, specifications, etc.

b. Task Assignment: assigning tasks to each member of the specifying team, and interfacing with the process monitoring service.

c. Work-plan and Schedule: planning the work for the participants of the process and setting a work schedule.

d. Document Distribution: distributing documents according to project mailing lists, e.g., distribution of raw requirements to users, and follow-up when responses are requested.

e. Error Processing: keeping track of errors that are revealed in the process and ensuring their proper correction.

f. Project Metrics: accumulating data and statistics about the project to be used for other projects and for estimations of the current project.

g. Cost and Time Modeling: obtaining the project statistics for models (see [Boe81]) that estimate the project's duration and cost.

12.17 Process monitor

The requirements specification process comprises several stages, each composed of several sub-stages that manipulate different files. The situation becomes more complicated because several stages may be reiterated, and because different persons of the specifying team can be in different stages at the same time.

The process monitor service directs the flow of the process and keeps track of where everyone is. Furthermore, it can suggest what the next stage should be, determine a pattern of action and warn when stages are overlooked.

CHAPTER 13
Conclusions

13.1 Method

The method for producing requirements was fully developed and presented. Some of the properties of this method are as follows:

- It is complete and exhaustive.
- It does not demand a prior knowledge about the specified system.
- It is user-oriented.
- It is independent of the application domain.
- It is independent of the way the requirements are formulated and stated.
- It is practical and easy to follow and implement.
- It is manageable for large projects.
- It provides several ways for system presentation.
- It can be partially automated.

A good way to rate a method is by its products. This can be done by obtaining principles for good specifications and analyzing the method according to them. One such set of principles good specifications is given by Balzer and Goldman [Bal81].

1. Functionality must be separated from implementation.

The process starts with users and their requirements. This usually ensures the required separation unless a user states his requirements in implementation terms. Scrutinizing by the specifiers during the raw requirements processing stage can eliminate implementation-oriented requirements.

2. The system specification language should be process-oriented.

The method provides several representations of the specified system. The operational system specification stage provides a representation based on basic functional operations and operational processes.

3. Specifications must encompass the system of which the software under design is a component

This principle is attained in two ways. One, the system of which the software is a component can be treated as an abstract user, decomposed into other objects that interact with the software. This interaction is stated as an abstract requirement. The second way is by the operational system specification stage as stated above.

4. Specifications must encompass the environment in which the system operate.

One of the presentations of the specified system is as a component of its environment. The operational system specification stage takes care of that.

5. The System specification must be a cognitive model.

This method describes the system as perceived by its user community. This perception constitutes the cognitive model.

6. Specifications must be operational.

This means that the specifications can be used to determine whether a proposed implementation satisfies them. This problem has not been addressed in the method.

7. The system specification must be insensitive to incompleteness.

The proposed method checks the requirements for completeness as perceived by the user and maintains requirements in several degrees of abstraction. New users and new requirements can easily be added due to the tree structure of the users domain.

8. Specifications must be localized and loosely coupled.

The proposed method yields loosely coupled requirements due to its processes. First requirements are generated by discrete users. Secondly, they are classified and sorted by discrete subjects. Both users and subjects are localized and loosely coupled. The exception is the class of general system requirements (See Chapter 10.) that are not localized.

The ultimate test of any process or methodology is out in the field, in real-life environment. This method has not been in the field used as a whole yet, but parts of it have been exposed to several large software systems. The projects were a logistic system, a command and control system, and a communication system. Each of them is estimated to be larger than 200,000 lines of code, and they all are still in development. Only initial, partial conclusions can be drawn presently.

- The existence of a method for specification, any method, was heartedly welcomed by the specifiers, project management and users.
- User identification for a large software system was a real problem for the specifiers. This problem is solved by the second stage of the method.
- The specifiers and the users value the manageability of the project caused by the method.
- Formal requirements were rejected by the implementors and the users. The use of SuperPDL [Bur83] with embedded natural language for requirements was successful. However, not all the users could understand and relate to requirements written in this way.
- Automation was requested by everybody. When the method was tested, no automated components

were available.

Some weaknesses were found

- The method seems to be too cumbersome for smaller projects. It does not adapt itself easily for scaling down.
- The management issues are not stressed enough.
- Automation is incomplete.

13.2 Automating the Process

The functional specifications for a computer-assisted specification system were described and the automation of each stage was discussed. Furthermore, a prototype of part of this system was implemented and used. The main conclusions are as follows,

- The method can be partially automated as a computer assisted system.
- The clerical part of the method and the management control can be fully automated.
- The reasoning, human activities can be partly automated as expert systems. Further research is needed here.
- The prototype has proved that it is possible to process requirements that are written in simple natural language, and that reasonable results can be gained by using limited techniques.
- Interaction between the computer and the specifier is essential.

A number of deficiencies were found.

- Only the composition, and to some extent the decomposition, processes were prototyped. The processes of finding classes, classification, and mapping were not tried out.
- The prototype is slow in dealing with large problems.
- It is not possible to maintain and process formal requirements with this system.

13.3 Recommendations for Further Research

Experimental

a. The whole process should be used extensively on real life projects and be evaluated and refined accordingly.

b. CAS should be used on several real and synthetic systems in order that it be better understood and evaluated. A reasonable knowledge base has to be built.

c. CASS should be developed and implemented. It should be distributed to several users in order that it be examined and evaluated.

d. More elaborate artificial intelligence techniques and ideas should be tested, e.g., skimming, natural language processing, expert systems, etc.

Theoretical

a. Develop a theory for the several representations of a system.

b. Devise algorithms for mapping, classification, and class formation.

c. Develop better tools to process requirements that are written in natural language.

CHAPTER 14
REFERENCES

[Abb83] Abbotts R.J., "Program Design by Informal English Descriptors", *Communications of the ACM* Vol. 26, No. 11, Nov 1983.

[Alf77] Alford M.W., "A Requirement Engineering Methodology for Real-Time Processing Requirements", *IEEE Transactions on Software Engineering* SE-3(1), 1977.

[ATL84] SuperPDL – An Interactive Software Design Tool., User's Manual, Advanced Technology Ltd., P.O.Box 13045 Tel-Aviv 61130 Israel.

[Bal79] Baltzer R., Goldman N., "Principles of Good Software Specifications and Their Implementation for Specification Languages", *Proceedings of the Conference on Specifications of Reliable Software*, Cambrige MA, April 1979.

[Bar82] Barr A., Feigenbaum A., *The Handbook of Artificial Intelligence*, HeurisTech Press, Stanford CA, 1982.

[Bas82] Ben-Bassat M., Dreizin Y., Carmon Y., Kashtan Y., "A Decision Support System for Construction and Maintenance of Developement Plans", Mekorot, Israel, 1981.

[Bei84] Beichter F.W., Hertzog O., Petzsch H., "SLAN-4 – A Software Specifications and Design Language", *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 2, Mar 1984.

[Bel76] Belford P.C., Bond A. F., Henderson D.G., Sellers L.S., "A Key to Effective Software Development", *Proceeding of the Second International Conference of Software Engineering*", San Francisco, CA, 1976.

[Bel77] Bell T.E., Bixler D.C., Dyer M.E. "An Extendible Approach to Computer-Aided Software Requirements Engineering" *IEEE Transactions on Software Engineering* SE-3(1), 1977.

[Bel79] Belady L.A., Lehman M. M. "The Characteristics of Large Software Systems", in *Research Directions in Software Technology*, Peter Wagner, Ed., Cambridge, MA: MIT Press.

[Ber83] Berry O., Berry D. M., "The Programmer-Client Interaction in Arriving at Program Specifications: A Methodology and Linguistic Requirements.", *Proceedings of the Conference on System Description Methodologies*, Kesckeemet, Hungary, May 1983.

[BeY83] Berry D. M., Yavne N., Yavne M., "On the Requirements for Program Design Language: Parametrization, Abstract Data Types, and Strong Typing in Software Design Process (SDP)", Computer Science Department, UCLA, 1983.

[Boe81] Boehm B. W., *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Boe83] Boehm B. W., Standish T. A., "Software Technology in the 1990's: Using an Evolutionary Paradigm", *Computer*, Nov 1983.

[Boe84] Boehm B. W., "Software Engineering Economics", *IEEE Transaction of Software Engineering*, Vol. SE-10, No., Jan 1984.

[Bon70] de Bono Edward, *Lateral Thinking*, Harper & Row Publishers, 1970.

[Bur83] Burstin M., Forscher Y., Maimon Y., Rotbard Y., "SuperPDL – A Software Design Tool" *IEEE 1983 SOFTFAIR*, 1983.

[Bur84] Burstin M., Ben-Bassat M., "A User's Approach to Requirements Analysis of a Large Software System", *ACM Conference*, Oct 1984 (To be published).

[Cai75] Caine S.H., Gordon E.K. "PDL - A Tool for Software Design" *Poceedings of the NCC*, 1975.

[Cas81] Casey, T., "Writing Requirements in English: A Natural Alternative" *IEEE Workshop on Software Engineering Standards*, San-Francisco, Aug 1981.

[Coh83] Cohen D., "Symbolic Execution of the GIST specification Language", *Proceedings of the Eighth IJCAI*, 1983.

[Cul78] Cullingford R. E. "Script Application: Computer Understanding of Newspaper Stories", 116, Yale University, Department of Computer Science, New Haven, Conn. 1978.

[Dav79] Davis A. M., Rauscher T.G., "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications", *Proceedings of Reliable Software*, Cambridge, MA, 1979.

[Dav82] Davis, "The Design of a Family of Application Oriented Requirement Languages", *Computer*, May 1982.

[DeJ79] DeJong G. F., "Skimming Stories in Real Time: An Experiment in Integrated Understanding", 158, Yale University, Department of Computer Science (1979), Ph.D. Dissertation.

[DeM78] De-Marco T., *Structured Analysis and System Specification*, New-York NY: Yourdon 1978.

[Est78] Estrin G., "A Methodology for Design of Digital Systems - Supported by SARA at the Age of One", *Proceedings of the National Computer Conference*, Anaheim CA, June 1981.

[Fis78] Fisher D. A., "DoD's Common Programming Language Effort", *Computer*, Vol. 11, Mar 1978.

[Gan79] Gane C., Sarson T., *Structured System Analysis* Englewood Cliffs NJ: Prentice-Hall 1979.

[Goo78] Good D.I. *et al.* "Report on the Language GYPSY - Version 2.0" Certifiable Minicomputer Project, University of Texas at Austin, Report ICSCA-CMP-10 Revision 1, 1978.

[Gut77] Guttag J., "Abstract Data Types and the Development of Data Structures", *Communications of the ACM*, June 1977.

[Ham76] Hamilton M., Zeldin S., "Higher Order Software - A Methodology for Defining Software", *IEEE Transactions on Software Engineering*, Mar 1976.

[Hei83] Heitmeyer C. L., McLean J. D., "Abstract Requirements Specifications: A New Approach and Its Applications", *IEEE Transactions of Software Engineering*, Vol. SE-9, No. 5, September 1983.

[Hic79] Hice G.F., Turner W.S., Cashwell L.F., *System Development Methodology*, North-Holland Publishing Co. 1979.

[HIP74] "HIPO - A Design aid and Documentation Technique", Form GC20-1851, IBM Corp., White Plans, N.Y., 1974.

[Hoa73] Hoare C.A.R., Wirth N., "An Axiomatic Definition of the Programming Language PASCAL" *Acta Informatica* 2, 1973.

[IEE84] IEEE Guide to Software Requirements Specifications, IEEE Std 830-1984, IEEE Inc., NY, USA, 1984.

[Kem80] Kemmerer, "FDM - A Specification and Verification Methodology", *Proceedings of the 3rd Seminar on DOD Computer Security Initiative Program*, National Bureau of Standards, Nov. 1980.

[Leh82] Lehman, M. M., "Program Evolution", Research Report DoC 82/1 , Dec 1982.

[Lev79] Levitt K.N., Robinson L., Siverberg, *The HDM Handbook* vol 1-3, Computer Science Laboratory, SRI International, Menlo-Park, June 1979.

[Lis75] Liskov B. H. and Zilles S. N., "Specification Techniques for Data Abstraction", *IEEE Transaction on Software Engineering* , March 1975.

[Lis79] Liskov B.H., Berzins V., "An Appraisal of Program Specifications", in Wegner P. Ed. *Research Directions in Software Engineering*, 1979

[Lud82] Ludewig J., "Computer Aided Specification of Process Control Systems", *Computer* , May 1982.

[Lud83] Ludewig J., "ESPRESO - A System for Process Control Software Specification", *Transactions of Software Engineering* vol. SE-9, No. 4, July 1983.

[Lun79] Lundeberg M., "An Approach for Involving the Users in the Specification of Information Systems", *Formal Models and Practical Tools for Information System Design* H.J. Schneider, Ed. North-Holland Publishing Co. 1979.

[Mus80] Musser D. R., "Abstract Data Type Specification in the AFFIRM System", *IEEE Transactions on Software Engineering*, vol. SE-6, Jan 1980.

[Mye78] Myers G. J., *Composite/ Structured Design*, Van Norstrand Reinhold Company 1978.

[Osb53] A.F. Oxborn, *Applied Imagination*, New York. Scribner, 1953.

[Par79] David L. Parnas "On the Criteria To Be Used in Decomposing Systems into modules", *Communications of the ACM*, December 1979.

[Ram83] Ramsey R. H., Atwood M. E., Van Doren J. R., "Flowcharts Versus Program Design Languages: An Experimental Comparison", *Communications of the ACM*, Vol. 26, No. 6, June 1983.

[Rob76] Robinson L., Levitte K.M., Neuman P.G., Saxena A.R. "A Methodology for the Design of Operating Systems Software" in: Yeh E.D. ed., *Current Trends in Programming Methodology*, Prentice-Hall 1977.

[Ros77] Ross D.T., "Structured Analysis (SA): A Language for Communicating Ideas", *IEEE Transactions on Software Engineering* SE-3(1), 1977.

[Rou76] Roubine O., Robinson L., "SPECIAL Reference Manual", Technical Report CGS-45. Menlo-Park, CA. SRI-International, 1976.

[Swa83] Swartout W. R., "The GIST Behavior Explainer", ISI Reprint series, ISI/RS-83-3, July 1983.

[SDM70] SDM/70 System Development Method. A proprietary product of Atlantic Software Inc., 1970.

[Sim77] Simon H.A., *The New Science of Management Decision*, Englewood Cliffs, N.J. Prentice-Hall 1977.

[Tei77] Teichrow D. Hershey iii E.A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", *IEEE Transactions on Software Engineering*, SE-3(1), 1977.

[Tho81] Thompson D. H., Gerhart S. L., Erickson R. W., Lee S., Bates R. L., eds., The AFFIRM Reference Library, 5 vols. USC Information Institute 1981.

[UNI83] UNIX Programmers Manual, Bell Telephone Laboratories Incorporated, Murray Hills, New Jersey, Holt, Rinehart and Winston, 1983.

[Van81] VanGundy A. B., *Techniques of Structural Problem Design*, Van Nostrand Reinhold Company, 1981.

[Was79] Wasserman A.I., "USE - A Methodology for the Design and Development of Interactive Information Systems" In: Schneider H.J. Ed. *Formal Models and Practical Tools for Information System Design*. North-Holland, 1979.

[Win82] Winchester J., Estrin G., "Requirements Definition and Its Interface to SARA Design Methodology for Computer Based Systems", *AFIPS Conference Proceedings*, Volume 51, June 1982, PP 369-379.

[Wit58] Whiting C.S., *Creative Thinking*, New York, Reinholds, 1958.

[Yeh77] Yeh R.T. *Current Trends in Programming Methodology* Vol 1, Prentice-Hall 1977.

[Zav82] Zave P., "An Operational Approach to Requirements Specifications for Embeded Systems" *IEEE Transactions on Software Engineering*, Vol SE-8, No. 3, May 1982.

APPENDIX A
The Products of the Requirements Production Method

A.1 Functional Specifications

The product of the functional specification stage is the first part of the final product. Its contents and organization are briefly discussed below. They follow the recommendations of the IEEE standard for requirements specifications SRS [IEE84]. The IEEE standard does not attempt to describe a methodology to arrive at the requirements. It merely prescribes the contents of the final requirements document. The output of the functional specification stage is a document that is organized following the table of contents in figure A-1.

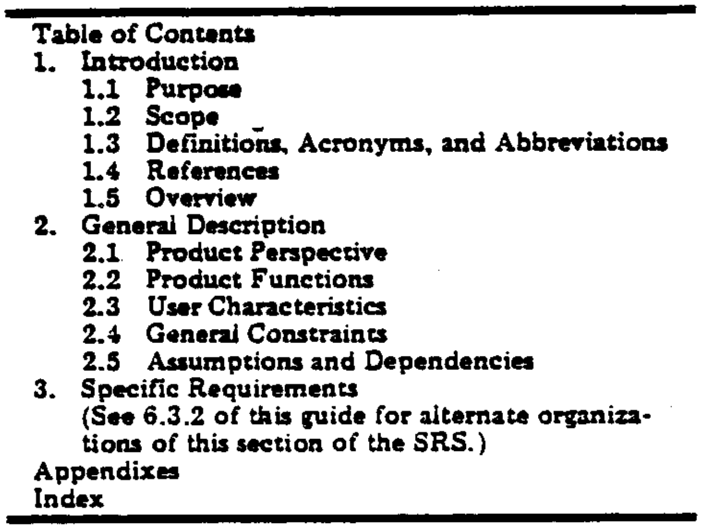
Table of Contents
1. Introduction
    1.1 Purpose
    1.2 Scope
    1.3 Definitions, Acronyms, and Abbreviations
    1.4 References
    1.5 Overview
2. General Description
    2.1 Product Perspective
    2.2 Product Functions
    2.3 User Characteristics
    2.4 General Constraints
    2.5 Assumptions and Dependencies
3. Specific Requirements
    (See 6.3.2 of this guide for alternate organizations of this section of the SRS.)

Appendixes
Index

Figure A-1: Prototype SRS Outline

Chapter 1, the introduction, provides the reader with the necessary background before going into details. It gives the purpose of the documents, their scope, the list of definitions and abbreviations, references to other documents that are used in preparing the SRS, and overview of the whole document. A list of action items to be taken by the readers of this document, mainly the users, is provided too.

Chapter 2 of the SRS document describes the general factors that affect the system and its requirements. It is divided into a number of sections as follows.

The Product Perspective Section relates the product to other systems or products.

The Product Functions Section provides a summary of all the functions that the software will per-

form.
The User Characteristics Section describes the general characteristics of the eventual users of the system.
The General Constraints Section provides a general description of other items that limit the developer's option for designing the system.
The Assumptions and Dependencies Section lists each of the factors that affects the requirements that are stated in the SRS.

Chapter 3 and the subsequent chapters are dedicated each to one of the subjects that comprises the whole system. Specifically, chapter 3 is dedicated to the general system requirements.

The following list describes the contents and structure of each chapter.

a. Sub-functions are represented in a tree structure, as parts of the main function.

b. The structure of each chapter follows the guidelines of the IEEE standard 830-1984 [IEE84]. Figure A-2 suggests the outline of a chapter.

c. The above mentioned outline lend itself to a hierarchical presentation. The presentation should be top-down.

d. A reference is given to indicate the source of any specification or requirement. The reference should map a specification to the corresponding raw requirements and to a user or to several users.

e. When a reference to a requirement is a scenario, then the scenario and the requirements that are derived from it should be indicated.

Chapter 4 lists the main recommendations that are derived from the specifications analysis and the preliminary recommendations for requirements that have been uttered by users, but should not be included in the designed system.

A.2 Operational System Specifications

A.2.1 Table of Contents

The product of this stage is the operational system specifications. This document is outlined here. The table of contents is:

3. Specific Requirements
3.1 Functional Requirement 1
3.1.1 Introduction
3.1.2 Inputs
3.1.3 Processing
3.1.4 Outputs
3.1.5 External Interfaces
3.1.5.1 User Interfaces
3.1.5.2 Hardware Interfaces
3.1.5.3 Software Interfaces
3.1.5.4 Communication Interfaces
3.1.6 Performance Requirements
3.1.7 Design Constraints
3.1.8 Attributes
3.1.8.1 Security
3.1.8.2 Maintainability
. . . .
3.1.9 Other Requirements
3.1.9.1 Data Base
3.1.9.2 Operations
3.1.9.3 Site Adaption
. . . .
3.2 Functional Requirement 2
. . . .
3.n Functional Requirement *n*

Figure A-2: Prototype Outline for SRS Chapter

7. Services and Users' Views
8. Logical Data Base
9. Back-Up and Recovery
10. Security
11. Documentation
12. Training
13. Organization and Operation
14. Operational Contribution

The first four chapters describe the environment in which the specified system operates, the next three chapters describe the operation of the system and the rest describe derived issues.

A.2.2 Introduction

This chapter provides the reader with necessary background information. Its sections are

1. General
2. The purpose of the document

3. Sources

4. Method

5. Contents

6. Action items

A.2.3 Purpose

This chapter presents the operational goals of the system and how compliance with these goals is to be checked.

1. Objectives and Goals

2. Criteria, i.e., the criteria for checking the system's goals

3. Measures, i.e., quantitative measures to be applied while checking for system goals

A.2.4 Principles and Constraints

This chapter defines the basic framework of the designed system.

1. Principles, i.e., operational principles upon which the design of the basic functions and constraints were based

2. Constraints, i.e., constraints on system operation

A.2.5 Interface to Other Systems

This chapter defines the borderline between the specified system and adjacent systems and the interfaces among them.

1. System Domain, which lists the subjects that are handled by the designed system

2. Adjacent Systems, which lists all other systems that can potentially have any interface with the designed system, and the suggested functional interface

A.2.6 Basic Operational Functions

These chapters detail every basic operational functions. A chapter is dedicated to each function.

1. General, i.e., a general description of the function, its goals and its role in the designed system

2. Description of the function

    1. Invocation
    2. Functional Operation
    3. Users

3. Operational Contribution, i.e., the benefit of this function to achieving the system's goals
4. Graphic Description

A.2.7 Operational Processes

These chapters describe the processes and how the designed system is involved with them. A chapter is dedicated to each process. An introductory chapter surveys all the processes For each process, the chapter has the following form.

1. General, i.e., a description of the process, its operational goal and its interaction with the designed system
2. The Present Process, a description of the process in the present (old) system, if such a process does exist
3. Description of the process
4. Graphic Description of the process
5. Operational Benefit, i.e., the benefits that the process provides for the operational goals of the designed system
6. Abstract Data Types, i.e., a list of the abstract data types that are involved in the process

A.2.8 Services and Users' Views

These chapters presents a list of all the system users and how the system is perceived by them. Here the information of the previous chapters is presented according to the user tree structure (See Chapter 5 of this thesis). A chapter or section is dedicated to each user, or to each abstract user in every level:

1. User Description
2. Organizational Structure
3. Required Changes in Organization Structure
4. System Services to the User

A.2.9 Logical Data Directory

This chapter lists the abstract data types that have been generated by the method. Most of the list is derived from the abstract requirements tree (See Chapter 8 of this thesis).

A.2.10 Back-Up and Recovery

This chapter lists the system requirements and the functional solution for system back-up and recovery. Most of these requirements are system level requirements.

A.2.11 Security

This chapter describes the security and privacy functions of the system and how they relate to the requirements in these areas.

A.2.12 Documentation

This chapter describes the functional documentation that is needed by each user for training and for operation.

A.2.13 Training

This chapter describes the training needs of each user before he can properly interact with the system.

A.2.14 Organization and Operation

Implementation of a new system frequently demands organizational and operational changes. This chapter identifies these changes, that are elaborated and detailed during the design phase.

1. Organizational Changes

2. Operational Changes

A.2.15 Operational Contribution

This chapter analyzes the contribution of the designed system to the organization or to the customer. This contribution is described by subject and by the operational goals of the system.

A.2.16 Cost Estimations

At this point the specifier may be able to come with first order estimated costs of implementing the system. The COCOMO model [Boe81] can be used here.

APPENDIX B
An Example: A Banking System

B.1 CAS Session

This is a a full session of CAS as used for the banking system (see figure 5-1). The actual CAS commands are given here. The results of the compositions of the natural language presentation and of the verb - object presentation are shown in the subsequent sections.

```
(1)  spc bank 'a banking system'
(2)  adu bank branch
(3)  adu . customers
(4)  adu . 'adjacent systems'
(5)  adu . management
(6)  adu branch 'branch manager'
(7)  adu . tellers
(8)  adu . back-office
(9)  adu customers drawers
(10) adu . loanees
(11) adu . depositors
(12) adu 'adjacent systems' communication
(13) adu . atm
(14) adu . accounting
(15) adu management president
(16) adu . 'executive vp'
(17) adu . control
(18) adu . 'internal audit'
(19) adu . fed
(20) adu loanees car
(21) adu . personal
(22) adu . business
(23) adu . mortgage
(24) adu depositors 'new account'
(25) adu . 'old account'
(26) adu . 'other branch'
(27) adu . 'other bank'
(28) adu accounting 'fixed assets'
(29) adu . 'accounts payable'
(30) adu . 'accounts receivable'
(31) adu . 'general ledger'
(32) adr 'new account' n1 check account
(33) adr . n1 open account
(34) adr . n2 write per-data
(35) adr . n3 write fin-data
(36) adr 'old account' o1 locate account
(37) adr . o2 compute balance
(38) adr . o3 write fin-data
```

(39) **adr 'other branch' br1 locate branch**
(40) **adr . br2 locate account**
(41) **adr . br3 compute balance**
(42) **adr . br4 write fin-data**
(43) **adr 'other bank' bk1 locate bank**
(44) **adr . bk2 send data**
(45) **adr mortgage m1 write appraisal-data**
(46) **adr . m2 write per-data**
(47) **adr . m3 check credit**
(48) **adr . m4 compute payments**
(49) **adr . m4 compute profitability**
(50) **adr . m5 issue loan**
(51) **adr business ba1 locate bus-data**
(52) **adr . ba2 check credit**
(53) **adr . ba3 check collateral**
(54) **adr . ba4 compute payments**
(55) **adr . ba5 compute profitability**
(56) **adr . ba6 issue loan**
(57) **adr personal p1 locate per-data**
(58) **adr . p2 check credit**
(59) **adr . p3 compute payments**
(60) **adr . p4 compute profitability**
(61) **adr . p5 issue loan**
(62) **adr car c1 locate car-data**
(63) **adr . c2 compute payments**
(64) **adr . c3 compute profitability**
(65) **adr . c4 issue loan**
(66) **cmr loanees ▾**
(67) **cmr loanees o**
(68) **usr loanees**
(69) **cmr depositors ▾**
(70) **cmr depositors o**
(71) **usr depositors**
(72) **cmr customers ▾**
(73) **cmr customers o**
(74) **usr customers**

B.2 Composition of Requirements - Free Format

Here the requirements are written in natural language under each elementary abstract user. Composed requirements of a higher level appear under the abstract user name which is indented.

Dep-New-Account
Check all account details.
Open new account by writting the necessary personal data.

Dep-Old-Account

Locate the account in the data base.
Compute the new balance and update the record accordingly.
Write the updated record.

Dep-Other-Branch
Identify data base to look at.
Locate record of account in the data base.
Compute the new balance and update the record accordingly.
Write the updated record.

Dep-Other-Bank
Communicate to the other bank or switch to a data base of transferred records.
Prepare data to be sent over.

Depositors

Locate the account whether it resides in a local data base or in a remote data base.
Check the appropriate details and compute the new balance.
Update and rewrite the record.
Invoke communication activities whenever necessary.

Loanee-Mortgage
Locate personal data about the loanee and appraisal data about the property.
Check the credit history of the loanee.
Check the equity of the property.
Compute the terms of the desired loan.
Compute the profitability of the transaction.
Issue the loan.

Loanee-Business
Locate the necessary data about the business and check its credit history.
Check the suggested collateral.
Compute the terms of the desired loan.
Compute the profitability of the transaction.
Issue the loan.

Loanee-Personal
Locate the necessary information on the person.
Check the credit history of the loanee.
Compute the terms of the desired loan.
Compute the profitability of the transaction.
Issue the loan.

Loanee-Car
Locate the necessary data about the car and about the loanee.
Check the equity of the car.
Compute the terms of the desired loan.
Compute the profitability of the transaction.

Issue the loan.

Loanees

Locate the data about the loanee and whenever necessary about the collateral and the equity of the subject of the loan.
Determine according to all the credit indicators if the loan should be granted.
Compute the terms of the desired loan.
Compute the profitability of the transaction.
Issue the loan.

Customers

Fetch customer data and other relevant data.
Check the relevant indicators to see if the transaction could take place.
Compute all the new customer and transaction parameters.
Compute the profitability whenever necessary.
Invoke other systems to perform special tasks.

B.3 Composition of Requirements - Function-Object Presentation

Here are the results of the composition done by CAS by using a knowledge base. The requirements are presented in the verb - object presentation.

Composition of Requirements: "Dep" to "Depositors"

Dep-New-Account (1.2.1.1)	Dep-Old-Account (1.2.1.2)	Dep-Other-Branch (1.2.1.3)	Dep-Other-Bank (1.2.1.4)
check account open account write per-data write fin-data	locate account compute balance write fin-data	locate branch, account compute balance write fin-data	locate bank send data
↓	↓	↓	↓

↓

locate account, branch, bank
open account
check account
send data
compute balance
write per-data, fin-data

↓

fetch account
send data
compute balance
write account-data

[per-data, fin-data → account-data
account, branch, bank → account
open, locate, check → fetch]

Composition of Requirements: "Loan" to "Loanees"

Loan-Mortgage (1.2.2.1)	Loan-Business (1.2.2.2)	Loan-Personal (1.2.2.3)	Loan-Car (1.2.2.4)
write appraisal-data	locate bus-data	locate per-data	locate car-data
write per-data	check credit	check credit	compute payments
check credit	check collateral	compute payments	compute profitability
compute payments	compute payments	compute profitability	issue loan
compute profitability	compute profitability	issue loan	
issue loan	issue loan		

↓ ↓ ↓ ↓

locate bus-data, per-data, car-data
write per-data, appraisal-data
check credit, collateral
compute payments, profitability
issue loan

↓

fetch loan-data
check credit-indicators
compute payments, profitability
issue loan

[locate, write → fetch
bus-data, per-data, car-data → loan-data
credit, collateral→ credit-indicators]

Composition of Requirements: "Loanees" and "Depositors" to "Customers"

Depositors (1.2.1)	Loanees (1.2.2)
fetch account	fetch loan-data
send data	check credit-indicators
compute balance	compute payments, profitability
write account-data	send loan

↓ ↓

Customers (1.2)
fetch account, loan-data
compute balance, payments, profitability
check credit-indicators
issue loan

↓ ↓

fetch customer-data
check credit-indicators
compute customer-parameters
compute transaction-parameters
issue loan

Consequences:

1. New requirement: compute profitability of an account.

2. New abstract user: Account Payable system.

APPENDIX C
Users Manual of CAS

CAS users manual is presented in the subsequent pages. For each command the manual presents its name, its synopsis, a description of what the command does, an example of the usage of the command, error messages that may be displayed during the use of the command, the list of files that are involved, the actual action that is taken by this command, other information that pertains to this command, and references to other commands.

NAME

spc – start a specification design

SYNOPSIS

spc *user-name user-description*

DESCRIPTION

spc initiates a specification design of CAS (Computer Assisted Specification). It should be the first command in a session.

user-name is the name of the specification design and the name of the root user.

user-description is the description of that user. The description parameter is optional, and is discarded when a design already exists (i.e., if the design has been initiated before).

EXAMPLE

1 Initiate a new specification design for a banking system

```
( )      spc      bank     'banking system users'

new specification session bank
USER NAME            USER DESCRIPTION
bank                 bank system users
```

2 Initiate an existing specification design for a command and control system

```
( ) spc cc
specification design cc exists
```

ERRORS

1 Specification design name is missing
No name was given for the session.

FILES

CAS000	all existing specification designs
CAS001	current specifications session
user-name.usr	user's file

ACTION

1 File CAS000 is searched for *user-name*. If it is found, it is copied to CAS001 and the program exits.

2 If it is not found, a directory by the name *user-name* is created in the home directory, and a file by the name *user-name*.usr is created in that directory. The file contains user's name and description.

MISCELLANEOUS

1 Users can be added to a root user only after the *spc* command.

2 To end a specification session use a *ends* command must be issued.

3 All the CAS commands start by referring to the file CAS001. If this file does not exist, an error message is issued.

4 CAS does not provide means to update CAS000 other than adding new specification design names. Use any editor when updating is needed.

SEE ALSO

ends, adu
Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University, Tel-Aviv, Israel.

NAME

ends − ends a specification session

SYNOPSIS

ends

DESCRIPTION

ends ends the current specification design session. A new *spc* command has to be initiated to begin the next session.

EXAMPLE

End the current session

( ) **ends**

FILES

CAS001	current session name

ACTION

The file CAS001 is removed.

SEE ALSO

spc

Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University, Tel-Aviv, Israel.

NAME

adu − add a user (child)

SYNOPSIS

adu *user-name* *new-name* *new-description*

DESCRIPTION

adu adds a child to the user *user-name*. This is the only command that allows introduction of new users to a system. New users can be introduced only as children to existing users.

The parameters *user-name* and *new-name* are parent's and child's names respectively. These are mandatory parameters. *New-description* is the description of the new user (child) and is optional.

EXAMPLE

Add a user called "management", who stands for the corporate bank management, to the user "bank".

```
( ) adu management 'corporate bank management'
new user management added to parent bank.

USER NAME           USER DESCRIPTION
management          corporate bank management
```

ERRORS

1 usage: adu user-name new-name new-description
At least one parameter is missing.

2 user *user-name* does not exist
An attempt was made to add a child to a non-existent user.

3 user name *user-name* already exists
An attempt was made to add a new user with a name that already exists in the specification design.

FILES

user-name.usr	user's file

ACTION

1 The session is initiated from CAS001.

2 If the number of parameters is less than two, an error message (1) is issued and the program exits.

3 *user-name* is searched for. If it does not exist the program issues an error message (2) and exits.

4 *new-name* is searched for. If it does not exist the program issues an error message (3) and exits.

5 *new-name* directory is built in *user-name*'s directory, and *new-name*.usr file is created in *new-name*'s directory.

6 The program declares a successful introduction of a new user.

MISCELLANEOUS

Please note that CAS does not allow two different users with the same name in the whole design.

SEE ALSO

adu, chu, usr
Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University, Tel-Aviv, Israel.

NAME

dlu − delete a user

SYNOPSIS

dlu *user-name*

DESCRIPTION

dlu deletes the user *user-name*. A user can be deleted only if it has no children. In order to delete a whole subtree, the users have to be deleted one by one bottom up.

EXAMPLE

Delete the user "management" from the specification design.

```
( ) dlu management
user management deleted from parent /bank
```

ERRORS

1 `usage: dlu user-name`
A *user-name* parameter is missing.

2 `user` *user-name* `does not exist`
An attempt was made to delete a non-existent user.

3 `user` *user-name* `has children and cannot be deleted`
An attempt was made to delete a user that has children.

4 `user` *user-name* `has requirements and cannot be deleted`
An attempt was made to delete a user that has requirements.

FILES

user-name.usr	user's file

ACTION

1 The session is initiated from CAS001.

2 If there is no parameter an error message (1) is issued and the program exits.

3 *user-name* is searched for. If it does not exist, the program issues an error message (2) and exits.

4 The directory of *user-name* is checked. If it has children (other directories), the program issues an error message (3) and exits.

5 The directory of *user-name* is checked for original requirements (*.req files) or composed requirements (r002). If there are any requirements the program issues an error message (4) and exits.

6 The user's file (*user-name*.usr) is removed, and the user's directory is removed.

MISCELLANEOUS

Requirements can be deleted using the *dlu* command.

SEE ALSO

adu, chu, usr
Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University, Tel-Aviv, Israel.

NAME

chu − change a user

SYNOPSIS

chu *user-name new-name new-description new-parent*

DESCRIPTION

chu changes user's data such as name and description. It can also change a user's parent. All new parameters are optional, but appearance of at least one of them is mandatory. Two apostrophes ('') means that a parameter does not appear and should not be changed.

Whenever the parameter *new-parent* appears, the whole subtree whose node is *user-name* is transferred to *new-parent*.

EXAMPLE

1 Change the user "bank-officers" to "bank-employees" with appropriate change of description:

```
( ) chu bank-officers bank-employees 'all the employees of the bank'

user  bank-officers changed

USER NAME           USER DESCRIPTION              PARENTS
bank employees      all the employees of the bank  bank/
```

2 Change the parent of the user "John" from "customers" to "management": Do not change the name or the description.

```
( ) chu John '' management

user John changed

USER NAME           USER DESCRIPTION              PARENTS
John                Mr. John Smith                management
```

3 Change the user "swords" to "ploughs", change the description and move him to the parent "agriculture" (former parent was "defense"):

```
( ) chu swords ploughs 'much better' agriculture
user swords changed

USER NAME           USER DESCRIPTION              PARENTS
swords              much better                   gov/agriculture
```

ERRORS

1 usage: chu user-name new-name new-description new-parent
At least one parameter is missing.

2 user name *user-name* does not exist
An attempt was made to update a non-existent user *user-name*.

3 new parent *new-parent* does not exist
An attempt was made to move a user to a non-existent parent.

FILES

user-name.usr	user's file

ACTION

1 The session is initiated from CAS001.

2 The number of parameters is checked. Error message (1) is displayed if necessary.

3 *user-name* is searched for. If it does not exist, the program issues an error message (2) and exits.

4 If *new-parent* is mentioned then *new-parent* user is searched for. If it does not exist, the program issues an error message (3) and exits.

5 The file *user-name*.usr is updated, and its name changed to *new-name*.usr.

6 If *new-parent* exists, then the directory *user-name* is moved to the directory *new-parent*.

7 The program displayed a message that describes the changes.

SEE ALSO

adu, dlu, usr

Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University, Tel-Aviv, Israel.

NAME

usr – display all the information pertinent to a user.

SYNOPSIS

usr *user-name*

DESCRIPTION

usr displays all the information pertinent to a user: its name and description, parent, children (if any), original and composed requirements (if any).

EXAMPLE

Display the information about the user "bank-officers":

( ) **usr bank-officers**

```
USER NAME         USER DESCRIPTION        PARENTS
bank-officers     officers of the bank    system/bank/

SONS
management
credit-officers
branch-managers
marketing-officers

no requirements
```

ERRORS

1 usage: usr user-name
No user name was given.

2 user *user-name* does not exist
An attempt was made to display a user that does not exist.

FILES

user-name.usr	user's file

ACTION

1 The session is initiated from CAS001.

2 If there is no parameter an error message (1) is displayed.

3 *user-name* is searched for. If it is not found, an error message (2) is displayed and the program exits.

4 User's name, description and parent is displayed.

5 If the user has children, their names are displayed.

6 If the user's directory contains original or composed requirements, they are displayed.

SEE ALSO

adu, dlu, chu, adr, dlr, mvr, cpr, chn, cmr
Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University, Tel-Aviv, Israel.

NAME

adr − add a requirement to a user

SYNOPSIS

adr *user-name req-name verb object text*

DESCRIPTION

adr attaches a requirement to the user *user-name*. The requirement is identified by a name *req-name* that is unique for a certain user. A requirement is stated by an affirmative sentence that contains a *verb* and an *object*. The verb and the object are processed by the composition program. The *text* is optional and is not further processed.

EXAMPLE

Attach a requirement "check account" to a user "data-base". The requirement name should be A.xx/003:

( ) adr data-base A.xx/003 'check account' 'check the account #'

```
new requirement A.xx/003 added to user data-base
number of requirements:
         4
REQ NAME    VERB        OBJECT              TEXT
A.xx/003    check       account       check the account#
```

ERRORS

1 usage: adr user-name req-name verb object text
At least one parameter is missing.

2 user *user-name* does not exist
An attempt was made to add a requirement to a user that does not exist.

3 requirement name *req-name* already exists for user *user-name*
User name *user-name* has already a requirement with the name *req-name*.

FILES

req-name.req	a requirement file

ACTION

1 The session is initiated from CAS001.

2 If there are fewer than four parameters, an error message (1) is displayed and the program exits.

3 *user-name* is searched for. If it is not found an error message (2) is displayed and the program exits.

4 *req-name* is searched for in the user's directory. If it does not exist an error message (3) is displayed and the program exits.

5 A requirement file by the name *req-name*.req that contains the requirement information (verb, object, text), is constructed in the user's directory.

6 A message that describes the change is displayed.

MISCELLANEOUS

1 Requirement name has to be unique for a whole design.

2 The requirement name is not preserved after composition.

SEE ALSO

usr, dlr, chr, mvr, cpr
Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University,

Tel-Aviv, Israel.

NAME

dlr – delete a requirement

SYNOPSIS

dlr *user-name req-name*

DESCRIPTION

dlr deletes a requirement *req-name* of the user *user-name*.

EXAMPLE

Delete the requirement name "C38.1" from the user "bank-customers":

```
( ) dlr  bank-customers C38.1
requirement C38.1 deleted from user bank-customers
number of requirements of user bank-customers:
        3.
```

ERRORS

1 usage: dlr user-name req-name
At least one parameter is missing.

2 user *user-name* does not exist
An attempt was made to delete a requirement from a non-existent user.

3 requirement *req-name* does not exist for user *user-name*
An attempt was made to delete a non-existent requirement from the user *user-name*.

FILES

req-name.req	requirement file

ACTION

1 The session is initiated from CAS001.

2 If there are fewer than two parameters then an error message (1) is displayed and the program exits.

3 *user-name* is searched for. If it does not exist an error message (2) is displayed and the program exits.

4 *req-name* is searched for in *user-name*'s directory. If it does not exist an error message (3) is displayed and the program exits.

5 The requirement file *req-name*.req is removed from the user's directory.

6 A message that describes the change is displayed.

SEE ALSO

usr, adr, chr, mvr, cpr
Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University, Tel-Aviv, Israel.

NAME

chr − change a requirement

SYNOPSIS

chr *user-name req-name new-verb new-object new-text*

DESCRIPTION

chr changes the contents of a requirement. Any of: verb, object, text can be changed. An empty field (two single quotes '', or no parameter when no other parameter follows) causes the corresponding field to remain unchanged.

EXAMPLE

1 User "accountant" has a requirement named "audit.007" that states:
audit main-file do it very carefully
Change the requirement to:
check all-files

```
( ) chr accountant audit.007 check all-files
requirement audit.007 of user accountant changed to

VERB        OBJECT          TEXT
check       all-files       do it very carefully
```

2 Change the text of the above requirement to "do it fast"

```
( ) chr ' ' 'do it fast'
requirement audit.007 of user accountant changed to

VERB        OBJECT          TEXT
check       all-files       do it fast
```

ERRORS

1 usage: chr user-name req-name new-verb new-object new-text
At least one parameter is missing.

2 user *user-name* does not exist
An attempt was made to change a requirement of a non-existent user.

3 requirement *req-name* of user *user-name* does not exist
An attempt was made to change a non-existent requirement of the user *user-name*.

FILES

req-name.req	a requirement file

ACTION

1 The session is initiated from CAS001.

2 If there are fewer than four parameters, an error message (1) is displayed and the program exits.

3 A search is made for the user *user-name*. If it does not exist an error message (2) is displayed and the program exits.

4 A search is made for the requirement *req-name* in the user's directory. If it does not exist an error message (3) is displayed and the program exits.

5 All the non-empty fields in the command line modify the corresponding fields in the requirement file.

6 A message that states the changes is displayed.

MISCELLANEOUS

To change of the name of a requirement or its user use the command "mvr".

SEE ALSO

usr, adr, dlr, mvr, cpr
Meir Burstin, *Requirements Analysis of Large Software Systems*, Ph.D. Thesis, Tel-Aviv University, Tel-Aviv, Israel.

NAME

mvr − move a requirement from one user to another with an optional change of the name of the requirement.

SYNOPSIS

mvr *user-name req-name new-user-name new-req-name*

DESCRIPTION

mvr moves a requirement *req-name* from the user *user-name* to another user *new-user-name*. The name of the requirement can optionally be changed to *new-req-name*.

EXAMPLE

Move the requirement 007 from user "bank1" to user "bank2". Change its name to 008:

```
( ) mvr bank1 007  bank2 008
requirement 007 moved from user bank1 to user bank2
requirement name was changed to 008
number of requirements of user bank2:
                3
```

ERRORS

1 usage: mvr user-name req-name new-user-name new-req-name
At least one parameter is missing.

2 user name *user-name* does not exist
An attempt was made to move a requirement from a non-existent user.

3 new user *new-user-name* does not exist
An attempt was made to move a requirement to a non-existent user.

4 requirement name *req-name* does not exist for user *user-name*
An attempt was made to move a requirement *req-name* that does not exist for user *user-name*.

5 requirement name *new-req-name* already exists
An attempt was made to change the name of a requirement to a new name that belongs to another requirement.

FILES

req-name.re	requirement file
new-name.req	new requirement file

ACTION

1 The session is initiated from CAS001.

2 The number of parameters is checked. If there are less than 4 parameters an error message (1) is displayed and the program exits.

3 *user-name* is searched for. If it is not found an error message (2) is displayed and the program exits.

4 *new-user-name* is searched for. If it is not found an error message (3) is displayed and the program exits.

5 The user's directory is searched for requirement field *req-name*.req. If it is not found an error message (4) is displayed and the program exits.

6 The new user's directory is searched for a requirement file: either *req-name*.req if there is no change of name or *new-req-name*.req if there is a change of name. If such a file exists an error message (5) is displayed and the program exits.

7 The file *req-name*.req is moved from *user-name* directory to *new-user-name* directory. A change of name is done if necessary.