AD-A196 198

DTIC FILE COPY

1

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE | READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER AFIT/CI/NR 88- 45	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A METHODOLOGY FOR SCENARIO-BASED REQUIREMENTS EXPLORATION		5. TYPE OF REPORT & PERIOD COVERED MS THESIS
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) HILLIARD B. HOLBROOK, III		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: UNIVERSITY OF FLORIDA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 1988
		13. NUMBER OF PAGES 103
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) AFIT/NR Wright-Patterson AFB OH 45433-6583		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

DTIC
ELECTE
AUG 0 4 1988
S D

D

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

SAME AS REPORT

18. SUPPLEMENTARY NOTES

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER [illegible] 13 July 88
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

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

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A METHODOLOGY FOR SCENARIO-BASED REQUIREMENTS EXPLORATION

By

HILLIARD B. HOLBROOK III

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1988

ACKNOWLEDGEMENTS

I would like to thank the good people and things who made this masterpiece possible. First and foremost, I would like to thank Dr. Stephen Thebaut for his involvement and commitment to this effort. I also wish to thank Andy, Lowell, and Mark for their ideas and feedback. Moreover, I would like to thank Jane and Belda-Beast for enduring my imposition on their day-to-day routine for the past two semesters. I am also grateful to my Macintosh for doing so many neat things and not breaking when the warranty expired. Finally, I must thank John Cleese for his portrayal of Basil Fawlty who served as my role model throughout.

TABLE OF CONTENTS

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A METHODOLOGY FOR SCENARIO-BASED REQUIREMENTS EXPLORATION

By

HILLIARD B. HOLBROOK III

April 1988

Chairman: Stephen M. Thebaut
Major Department: Computer and Information Sciences

This thesis proposes a methodology for conducting requirements exploration based on the use of scenarios as a means for communication between a software system's user and designer. Requirements exploration is the process of determining what functions a user expects from a system and how the system should behave as it performs those functions. The effectiveness of this process has a large impact on the ultimate success of a software system.

The proposed methodology reflects several key features associated with effective requirements exploration. The first of these is the parallel and iterative development of a high-level design along with a set of system requirements. Because complex software systems are wicked problems, their requirements must be developed in light of possible solutions. A second feature is the use of scenarios as a basis for communication between users and designers. Due to their flexibility and informality, scenarios overcome some of the barriers associated with user-designer dialogs. A third feature involves users and designers together considering provisional designs. Such a process tends to uncover unstated, or clarify misunderstood, requirements. Finally, the methodology provides for an explicit linking between components of the design and the requirements that they satisfy. This ensures that attention is paid to all aspects of a problem as a design is formulated.

To support this methodology, considerations for a hypertext-based tool are also described. In general, hypertext systems provide a means for processing non-linear text through the use of machine-supported links. This capability supports the linking which is a fundamental component of the methodology's conceptual architecture. More specifically, a mapping of this architecture into Apple's hypertext system, HyperCard™, is described.

CHAPTER 1
INTRODUCTION

The hardest single part of building a software system is deciding precisely what to build. . . . No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

- Fred Brooks [Br87 p.17]

This thesis addresses the general problem of determining what to build when beginning a new software system. This introductory chapter provides an overview of the problem, research area, and objectives of this thesis. The first section describes the general problem motivating the research into requirements identification. The second section explores the research itself and its objectives. In the final section, an overview of the thesis is presented.

Requirements, Communication, and the Software Problem

In this introductory section, we will examine the role that inadequate requirements definition and ineffective communication play in what has become known as the *software problem*. The purpose is to highlight the need for research and improvements in the requirements identification process. We will begin by first using some dismal statistics to describe the software problem and then define requirements identification. From that point, we will explore the relationship between requirements identification, poor communication, and the software problem.

The Software Problem

"Most projects are completed significantly over budget, require more effort than estimated, and are completed late" [Je84 p.81]. This is a conclusion reached by Jenkins et al. in a study of seventy-two system development projects in twenty-three major U.S. corporations. Although Jenkins concludes that users are "generally satisfied with their systems" [Je84 p.81], another source suggests otherwise. At the 1982 DMPA software management conference, it was indicated that 25% of software development is never delivered, and 47% is delivered but not used [Gl82].

These figures all illustrate a "syndrome euphemistically called the *software problem*" [Tu87 p.2]. Belady described its scope as, "universal, not only from IBM out to the other mainframe companies, but also out to the aerospace companies. I found out that all of them had significant software problems and had several thousand professional programmers working for them" [My85 p.69].

Requirements Identification

Requirements analysis is a process in which "what is to be done" is elicited and modeled. This process has to deal with different viewpoints, and it uses a combination of methods, tools, and actors. The product of this process is a model, from which a document, called requirements, is produced. [Le87 p.26]

The above definition comes from a survey of requirements analysis authored by Leite [Le87]. In it, he points out that there is a great deal of confusion concerning the product and the process of requirements analysis. The result he says, is the eternal debate over what the difference between requirements and specifications is in terms of their use later in the project. Leite resolves the issue by determining whether or not the process in question addresses the topic of *requirements elicitation.*

Elicitation is a term used to describe the processes related to understanding, finding and gathering information. It should also consider the task of unfolding the tacit knowledge, and the

> communication process between users and analysts. . . . Elicitation can be seen as consisting of three components: fact-finding, experimentation, and communication. [Le87 p.28]

For our purposes, the terms requirements elicitation, requirements exploration, and requirements identification are defined by the above definition and used interchangeably.

The Role of Requirements Identification in the Software Problem

What role does the requirements identification process play in the software problem? As observed by many authors, if done incorrectly, it is a major culprit. Scharer noted: "One of the most common reasons systems fail is because the definition of system requirements is bad" [Sc81 p.139]. Jenkins et al. found that 65% of the projects they reviewed suffered to some degree from faulty or incomplete requirements [Je84]. Moreover, they found that 78% of the project leaders reported additional requirements being discovered after approval of the requirements statement, only 37% of which were treated as new development requests [Je84]. Finally, according to a study by the Savant Institute, 56% of all errors come from the requirements analysis phase of development [An83 p.17].

How is it that missed requirements can be so devastating? A nasty characteristic of missed requirements is that they tend to go unrecognized in a traditional development effort for a long time. This is coupled with the "observable fact that the longer errors remain undiscovered in the software, the more costly they are to remedy" [Ho82 p.88]. The net effect is that unarticulated requirements take a long time to recognize, and are therefore very expensive to fix.

The Role of Communication in the Software Problem

Communication plays an enormous role in requirements identification. Weaver defined communication "to include all of the procedures by which one mind can affect another" [We49 p.15]. During requirements identification, effective communication between users and designers has taken place when they share the same vision of a problem and what must be done to solve it.

Unfortunately, effective communication between users and designers is very difficult to achieve. In fact, a Delphi survey of programming managers revealed that communication between clients and designers were the largest source of problems in software [Sc74]. Malhotra notes: "such studies indicate that a better method of communicating goals from the ultimate users or buyers of programs and the software designer could *very significantly* impact the cost of software" [Ma80 p.133].

Research Objectives

The preceding section clearly shows the need for research in, and improvements to the requirements exploration process and the communication that takes place therein. This section outlines the research objectives associated with this thesis.

Upstream Activities

According to Belady [My85], the software development process begins with a world of fuzzy ideas, and ends with a world of bits and bytes. Belady divides the process at the point where there exists a relatively well formalized set of system specifications. Referring to the well-known waterfall diagram, *upstream activities* are characterized by a set of vague, intuitive mental processes that involve transforming the fuzzy ideas into relatively well formalized specifications. On the other hand, *downstream activities* involve more formal activities such as specification techniques, languages, compilers, testing, and verification.

Up to this point, there has been relatively little research into the upstream activities. Instead, computer science has focused on the downstream activities. Sadly, without an equal emphasis on the upstream activities, software developers may in fact be solving the wrong problems. The effect here is aptly described by von Neumann: "there's no sense in being precise about something when you don't even know what you're talking about" [Le87 p.3].

The importance of research into upstream activities and requirements exploration in particular is underscored by Brooks: "the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements" [Br87 p.17]. He goes on to say: "one of the most promising of the current technological efforts. . . is the development of approaches and tools for rapid prototyping of systems as prototyping is part of the iterative specification of requirements" [Br87 p.17].

Rapid Prototyping

As a response to the inability of the traditional method (a.k.a. the waterfall method) of software development to adequately support requirements analysis, rapid prototyping has evolved as a promising alternative. Prototyping involves quickly developing a version of the software that provides the basic functionality of the final system without being bound by the same hardware, size and performance constraints [Br87].

Although there has been a fair amount of research done on rapid prototyping[1], there are no well-accepted methodologies for its use. Accordingly, there are many views towards defining and applying prototypes. It has been found that prototypes serve as an extremely effective basis for communication between users and designers [Go83]. Moreover, their use has provided for more successful systems in terms of user acceptance

[1]Turner's thesis [Tu87] provides a good overview on rapid prototyping.

[Bo84]. Unfortunately, prototypes have also proven to be hard to manage and a very expensive way to explore requirements [Ho82].

Scenario-Based Requirements Exploration

It has been suggested that *scenarios* may offer an inexpensive alternative to prototypes in terms of expressing the behavioral characteristics of a proposed system [Ho82, We87]. Scenarios can be thought of as operational examples of a hypothetical system's behavior as experienced by a user. Simply stated, this area of research aims to provide a quick, inexpensive and effective way of conducting requirements exploration based on the use of scenarios as a point of focus for communication between the user and designer. The benefit is to provide software developers with a fast, effective, and cheap way of assessing provisional solutions to a user's problem and uncovering unstated requirements in the process. We henceforth refer to this process as *scenario-based requirements exploration.*

Towards this goal, this thesis provides a general background of the theory and process of software development with respect to requirements exploration. Based on this material, a high level architecture and methodology are described for conducting scenario-based requirements exploration. A general structure for developing tools to support the scenario-based requirements exploration process will be provided, and the methodology described will serve as basis for empirical research to determine the applicability of a scenario-based approach in various situations.

Associated with the proposed methodology are advantages related to the design process in general. Perhaps the most important of these is the incorporation of design into the requirements exploration process. Furthermore, the methodology provides for explicit linking between specific requirements and features of the design. Finally, the methodology provides a structure that facilitates the consideration of all of the requirements as a design is generated.

Organization of the Thesis

Chapter 2 is a background study that serves as a basis for the objectives and components of the methodology. In it, the principles and process of software development are explored with respect to identifying requirements. The first section concerns cognitive processes involved in software development including communication, problem solving, and design. The second section concerns present software development practices and includes an overview of requirement determination strategies. The final section investigates the use of scenarios as a flexible and informal method of communication in the design process. Conclusions drawn from this chapter serve as a basis for the methodology presented in the following chapter.

Chapter 3 describes a methodology for scenario-based requirements exploration. The first section describes a conceptual architecture for the types of information captured and manipulated during requirements exploration. The second section explains the process as an evolution of information in the architecture. Related to this material, Appendix A presents a hypothetical example of the use of the methodology.

Chapter 4 explores the applicability of a hypertext-based tool to support the methodology. The first section provides an overview of hypertext in general as it relates to the characteristics of the methodology. The next section offers a brief overview of HyperCard and describes how the conceptual architecture can be constructed in HyperCard. Appendix B describes in some detail, the implementation of such a system. Finally, Chapter 5 concludes the thesis with a summary of the ideas presented, an assessment of the methodology and a discussion of further research.

CHAPTER 2
BACKGROUND

In this chapter, background material relevant to requirements exploration is presented. First, the cognitive processes involved in software development are explored and discussed. From there, two ways in which software is developed, the waterfall method and prototyping, are explored along with their associated advantages and drawbacks. Finally, we will take a look at the role scenarios can play in software development.

Cognitive Processes in Software Development

The upstream software development activities involve various cognitive processes, the effectiveness of which determine, to a large degree, the effectiveness of systems development in general. In this chapter, the processes of communication, problem solving and design are discussed. From this discussion are derived the objectives and components of a requirements exploration methodology presented in the next chapter.

Communication

The communication process plays a major role in nearly every phase of software development, but particularly in requirements exploration. Steele and Nowell stress this importance:

> During needs determination, effective communication between information system users and information systems designers is critical--creative exploration of the problem environment is imperative. . . . The immediate usability of newly implemented information systems is directly proportional to the quality of cognitive creativity. [St83 p.226]

Further emphasizing the importance of communication, a Delphi survey of programming managers pointed to communication problems as the biggest source of problems in software [Sc74]. Consequently, any improvements in the communication process could very significantly impact total system cost. To examine the communication process, we will now explore a mathematical model for the communication process and discuss some hindrances to effective communication.

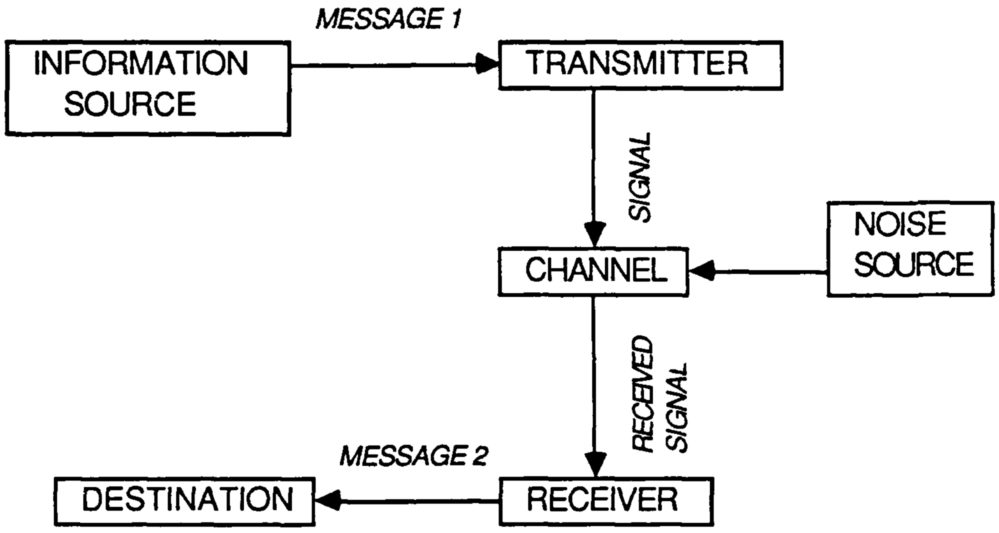


Figure 2-1. Weaver's Mathematical Model of Communication [We49]

The Communication process - a mathematical model

In the simplest sense, the communication process involves three entities; the originator, the signal, and the recipient. Communication takes place when the originator transmits a signal which is then received and interpreted by the recipient [Bo71]. Although this simple model can be further specified in any one of a number of other models, Weaver's mathematical model of communication [We49] is suitable for explaining station to station, one way communication. In 1949, Warren Weaver developed a mathematical model for communication as his analysis of the importance of Claude Shannon's work on the mathematics of information theory [Bo71].

Weaver defined communication "to include all of the procedures by which one mind can affect another" [We49 p.11]. The components of his model, shown in Figure 2-1, are explained as follows. A *message* is formulated at the *information source* (i.e., the mind of the originator) to be sent to the *destination* (i.e., the recipient). To do so, the originator encodes the message to be sent and employs a *transmitter* (e.g. a mouth, pencil, or keyboard) to change his message into a *signal* which is sent over the *channel* to the recipient. The recipient acts as a reverse transmitter, translating the received signal back into a message. Effective communication occurs when the message received matches the message sent by the information source.

A very important element of the model related to the effectiveness of communication is *noise*. Weaver defined noise as "certain things not intended by the information source added to the signal" [We49]. The effect of noise for the most part is to distort the signal in such a way that when it is received and decoded, it may not be the message intended.

Barriers to communication

Unfortunately, there are several influences that turn information into noise. These are described by Orrin Klapp in his book on the quality of life in our information society [Kl86]. Disrupting influences most relevant to the design process are decoding difficulties, bad complexity, sheer overload, and the dearth of feedback.

Decoding difficulties involve excessive effort on the part of the receiver. One of the main culprits in decoding difficulties is the use of jargon. According to Klapp, "our world is filled with jargons that baffle understanding. . . . While jargon may serve specialists, it makes language noisier--and more boring--to almost everybody else" [Kl86 p.87]. Rather than enhancing communication, jargon may serve as a proccupying cipher to the receiver.

In addition to decoding, the receiver is burdened with organizing the messages that he receives. Usually, a degree of complexity is more desireable than simplicity. As Klapp describes it, "good complexity has a pattern that is intelligible and often pleasing to the human mind--a signal that we can hope to understand and manage" [Kl86 p.91]. Alternatively, *bad complexity* is characterized by an element of confusion on the part of the receiver because no pattern is presented to serve as a key for organizing the messages.

A third barrier discussed by Klapp is sheer overload. It is an overload of channel capacity. As described earlier, channels carry the signals to the receiver. Individuals vary in their capacity to handle information. Accordingly, when the rate or amount of information exceeds a receiver's capacity, information becomes noiselike.

Finally, and perhaps most important from a designers point of view is what Klapp calls the dearth of feedback. As he put it, "lacking news of where one succeeded or failed puts a person in the predicament of the king who wore no clothes, unable to see what is wrong and rectify performance. Large volumes of information without feedback doesn't solve problems, but adds to the difficulty of finding fact or meaning" [Kl86 p.92].

Summary of communication

To briefly summarize, communication is a critical cognitive process involved in requirements exploration as well as the entire software development process. Consequently, there is great potential payoff for improvements in communications, especially between users and designers. A mathematical model of the communication process was discussed. One of the components of this model is noise, the effect of which is to distort messages. Relating to requirements exploration were four barriers to communication which affect noise: decoding difficulties, bad complexity, sheer overload, and the dearth of feedback.

Problem Solving

Vitalari and Dickson defined problem solving in systems analysis as "the reasoning process the analyst uses to analyze an information requirements determination problem and to synthesize a solution" [Vi83 p.949]. In this section, problem solving as it relates to requirements exploration is discussed. First, Newell and Simon's general theory [Ne72] is briefly reviewed, and then two related types of problems, wicked and ill-structured, are considered.

General problem solving theory

In order to advance our understanding of how humans think, Newell and Simon assembled several decades of work into a general theory of human problem solving [Ne72]. The theory applies to information processing systems in general, of which humans are a specific case. It is based on two fundamental concepts, the *task environment* and the *problem space.*

The task environment is the environment within the real world that is coupled with a goal, problem, or task. Formulated within limits imposed by the task environment, the problem space is composed of abstractions of the task environment. The problem space represents where the problem is solved and the way in which a decision maker chooses to work on the problem. Among the components within the problem space are: states of knowledge about the task, operators or information processes producing new states, the initial state of knowledge, and the desired goal state. In short, the problem space contains the total knowledge available to solve the problem, including the knowledge and experience of the solver.

Effective problem solving is associated with the ability of the problem solver to extract and exploit information from the task environment within the problem space. Newell and Simon noted that "the effectiveness of a problem solving scheme depends wholly on its reflecting aspects of the structure of the task environment" [Ne72 p.824]. Since humans have a limited capacity for dealing with complex problems, they employ a concept related to the problem space known as *bounded rationality*. Bounded rationality involves constructing simplifications in order to deal with complexity. These simplifications are usually in the form of simplified models of the task domain. In a study by Vitalari and Dickson, proficient systems analysts were found to use a general model to bound the problem space and aid in the efficient search for requirements [Vi83].

Wicked and ill-structured problems

With respect to the general theory, we will now examine some features associated with software design problems. Design problems share characteristics with two classes of problems known as *ill-structured* and *wicked* problems. Accordingly, design problems must be approached and evaluated somewhat differently than more well defined problems.

Characteristics of analysis problems.

Vitalari and Dickson listed six characteristics of the analysis task domain with respect to problem solving [Vi83].

1. At the inception, there exist ill-defined boundaries, structure, and a degree of uncertainty about the nature and make up of the solution.
2. The solutions to analysis problems are artificial in that they are designed and many potential solutions exist for any one problem.
3. Analysis problems change as they are being solved due to the organizational context and multiple participants involved in the specification process.
4. Solutions require interdisciplinary skill and knowledge.
5. The knowledge base of analysts is continually evolving so the analyst must be ready to adapt to changes in technology as well as different ways of interacting with users.

6. The process of analysis is primarily cognitive in nature requiring the analyst to structure the problem, process diverse information, and develop logical and consistent specifications. All other skills such as interpersonal interaction and organizational skill facilitate this cognitive process.

These characteristics relate closely to two types of problems known as *wicked* problems and *ill-structured* problems which we will now explore.

Wicked problems. Rittel and Webber coined the term *wicked problems* to describe problems such as social planning efforts [Ri73]. For example, in a pluralistic society, they noted that there are no objective criteria for concepts such as equity. Wicked problems contrast with tame problems in that tame problems have a clear mission and therefore a criteria with which a solution can be judged.

Accordingly, wicked problems cannot be precisely defined, and beyond that, cannot be evaluated in any objective sense. Guindon noted that many design problems, especially for novel applications, share the following subset of the characteristics with wicked problems [Gu87]:

1. Wicked problems lack a single correct formulation and every formulation of a wicked problem corresponds to a possible solution.
2. There is no stopping rule, no single set of properties that describe when a solution (goal state) is reached - one can always reach a better solution so resource limits serve as the stopping rule.
3. There is no exhaustive set of possible operators to be used at different stages of the solution process.

Conklin and Richter summarize the implications of viewing design problems as wicked problems as follows

> Because large scale software design is a wicked problem, the design process itself must be very complex. The formulation of problems, the identification of conceivable solutions (some based on previous, similar

> problems), and the evaluation of solutions in the absence of well-defined stopping rules, must all be significant and complicated components of any successful design process. Furthermore, because, as Rittel and Webber noted, the formulation (i.e. understanding) of the problem requires some knowledge of the conceivable solutions, these design activities must all occur concurrently, and not in separate, distinct phases. [Co85 p.2]

Ill-structured problems. Closely related, perhaps synonymous, to wicked problems are *ill-structured* problems discussed by Simon [Si73]. Ill-structured problems are defined as problems whose structures lack definition. The design process is ill-structured in two major respects. First, there is no definite criteria with which to test a solution. Second, the problem space is not defined in any meaningful way. To illustrate problem solving in design, Simon discussed the process employed by an architect to design a structure. He points out that we are referring to a creative process here and not the situation in which the architect simply selects an "off the shelf" design.

The architect's process. Initially, through discussion with the customer, the architect has some user-specified constraints and an incompletely specified set of design goals. From these incomplete goals, the architect formulates some global specifications. Although he evokes a guiding organization and specific attributes from memory, at no time in the process do they provide a complete procedure or information to design a house. Rather, more well-defined subgoals are addressed and the resulting interrelations of the subsolutions are incorporated into an overall design. With this approach, there are dangers of inconsistencies among the subsolutions. However, these dangers are tempered by the architect's skill in organizing his program for design [Si73].

Generally speaking, the architect's problem can be described as well structured in the small, but ill structured in the large. As Simon says, "the whole design, then, begins to acquire structure by being decomposed into various problems of component design, and by evoking, as the design progresses, all kinds of requirements to be applied in testing the design of its components" [Si73 p.190].

Serendipitous problem solving. As part of Microelectronics and Computer Techonolgy Corporation's (MCC) investigation of upstream activites, Guindon et al. conducted a verbal protocol study of professional software designers individually designing an n-lift elevator system [Gu87]. From the study, they identified what they considered to be the main sources of knowledge and processes which underlie the design control strategies. Additionally, they provided some general observations of the designers activities.

It was observed that a top-down structured approach was used only when a solution could be formulated immediately from the set of requirements, more novel situations required a higher degree of exploration. They observed an interesting problem solving approach to novel problems, one they call *serendipitous* problem solving. While not purely bottom-up, it involves approaching a novel problem by moving between different levels of abstraction and detail, motivated by the recognition of partial solutions.

Coordination of the partial solutions can be a problem with serendipitous problem solving. There is a tendency to forget to return to postponed subproblems. This was also noted in Simon's architect's process as "interrelations among the well-structured subproblems are likely to be neglected or underemphasized" [Si73 p.191].

Summary of problem solving

Our discussion of the problem solving process began with an overview of Newell and Simon's general theory. We presented the concepts of task environment and problem space, as well as a brief look at effective problem solving. Following this, we discussed two types of problems that share characteristics with design termed wicked and ill-structured problems. Finally, two approaches to these problems, serendipitous problem solving and the architect's process, were examined.

Design

As we have already seen, the design process is a specific kind of problem solving. According to Malhotra [Ma80], design problems are characterized by a lack of a specific initial state and fuzzy goals that can not be mapped directly onto the design properties. However, part of the design process involves formalizing and refining the requirements so that they can be matched by the transformations (or system design). In this section, we explore three models of the design process: Amkreutz' cybernetic model, Malhotra's model of the cognitive processes in design, and Sasso and McVay's constraint/assumption model.

Cybernetic model of the design process

In 1975, J. H. A. E. Amkreutz presented a cybernetic model of the design process [Am76]. Cybernetics is the theory of information processing dealing with the input, transformation, and output of information. Consequently, a cybernetic model treats information as a physical quantity apart from its carrier. Amkreutz' motivation for developing the model was to form a basis for developing computer-aided design systems which would provide for a better way of structuring, communicating, and processing the information involved with the design process.

At the highest level, shown in Figure 2-2, the cybernetic model can be viewed as a black box where input information *Ip* contains all of the relevant task related information such as requirements and constraints. It is input into the design process *P*, producing an output design *Op*. In broadest terms, the goal of the design process is to achieve an equilibrium in which all of the requirements and constraints in *Ip* are satisfied by the ultimate design *Op*.

To achieve this equilibrium, a feedback function operates within the design process as illustrated in Figure 2-3. Here, a feedback function *F* evaluates the design generated in the design generation function *G* using the criteria contained in the current input information *Ip* and regulates the information flow *Ig* back into G.

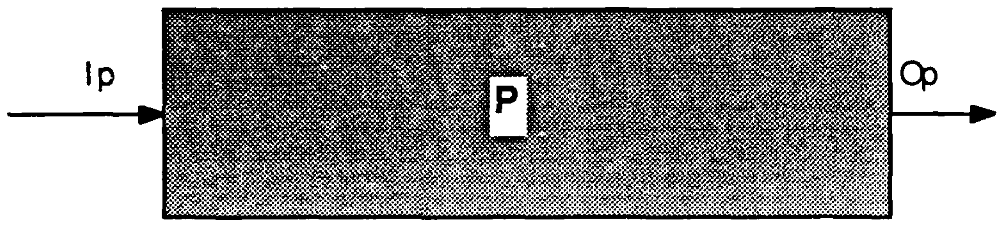


Figure 2-2. Black box model of the design process [Am76 p.188]

A further refinement of the feedback function, which Amkreutz calls the regulation function, is illustrated in Figure 2-4. Here, an evaluation function *E*, consists of a memory function and a function that determines the deviation of the relevant parameters in the design generation output, *Og*. Based on the nature of the deviation, a decision function *D* determines how *Ig* is to be changed and passes this information to *R*, the regulation function. The regulation function implements the strategy selected and makes the modifications to *Ig*.

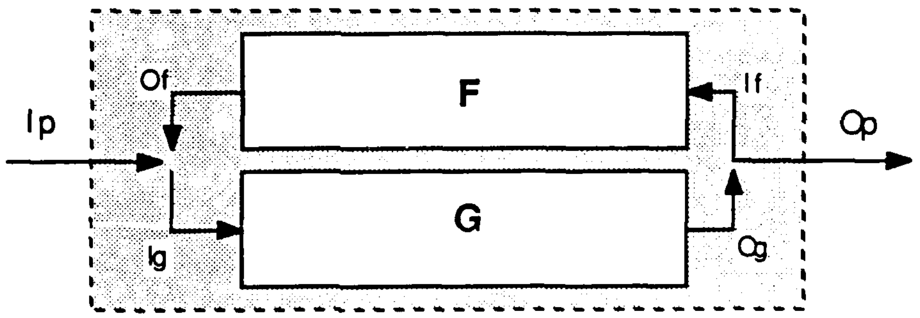


Figure 2-3. Design process as a control function [Am76 p.189]

The model has two features that are relevant to our purposes. First is the view that "preparation of design information, problem analysis, preliminary design and detailed design are integrated parts of the same process" [Am76 p.190]. In other words, the design process is not sequential and the formulation of the goals is done in parallel with the design. The second important feature is the description of a feedback function that evaluates and modifies the design.

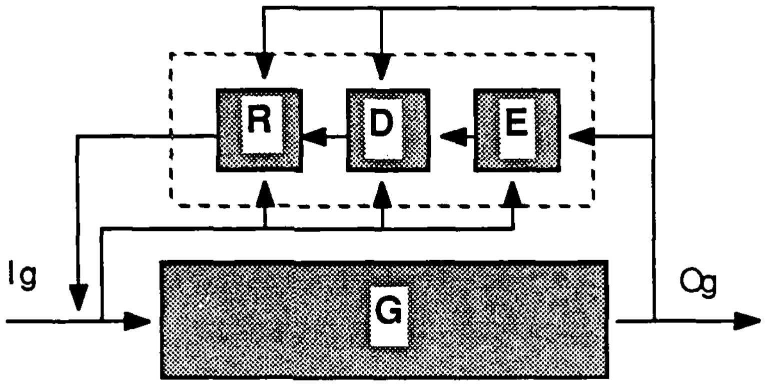


Figure 2-4. Regulation function [Am76 p.190]

Malhotra, Thomas, Carrol, and Miller

At IBM's Watson Research Center, Malhotra, Thomas, Carroll, and Miller studied the design process in an effort to improve the design of computer software [Ma80]. Specifically, they were interested in the processes of articulation of requirements and the generation and evaluation of subsolutions. They conducted a study of the design process by videotaping and transcribing actual client-designer dialogs. In addition to developing a model of design, they made the important observation that new and/or unstated requirements were uncovered as clients and users evaluated solutions. Significantly, they draw conclusions similar to Amkreutz' discussed in the previous section. They state:

> The goal elaboration process and the design generation process coexist within the design process. They are usually inextricably interlaced and

> they assist and strengthen each other; one fading into the other only to rise again a little later. Design evaluation occurs, as required, within the design generation process. [Ma80 p.133]

Based on their studies, a model of the design process was developed that involves a cyclic iteration of three processes; goal elaboration, design generation and design evaluation. The process is initiated by a user with a perceived need who attempts to formulate and articulate this need into specific goals. Here, a designer with a detailed knowledge of the design discipline is employed to help the user articulate his goals. The process of statement and discussion of goals is called *goal elaboration*. In effect, it involves the decomposition and selection of subgoals to a point where they are specific enough to be considered functional requirements.

When the goals have been decomposed, the second process, *design generation*, in which the designer formulates a design (or designs) that meets the functional requirements specified in the goal elaboration process begins. When a design (or partial design) is generated, *design evaluation* begins. This phase consists of the user and designer discussing the various properties of a design and how well it satisfies the stated goals. If a design is deemed satisfactory, it is accepted as a solution. However, during this process, new or modified goals may be formulated which will initiate another cycle of goal elaboration, design generation, and design evaluation. The cycle stops when all subgoals of the problem decomposition have been addressed and accepted during the design evaluation.

To summarize, the model presented by Malhotra is significant in several respects. Perhaps most important is the observation that new or unstated requirements are uncovered in the process of evaluating provisional solutions. Also important is the observation that the formulation of goals and design are in fact intertwined parts of the same process.

The Constraints/Assumptions Model

Sasso and McVay described a model of design that overcame many of the weaknesses they found with previous models. Two of the features of their model are important to us. First is the existance of links between the requirements and the aspects of the design that satisfy those requirements. Second is the iterative nature of design which they describe as a major component of the design process.

As the name implies, there are two basic components of this model, *constraints* and *assumptions*. Constraints serve to define the limits of the design space and assumptions are generated during the design process about the nature of the problem and its solution. To form a strong link between the problem definition and the solution, these assumptions must be explicitly stated and agreed upon by the user and designer. As these assumptions are verified by the user, it is expected that constraints will be redefined. Furthermore, the evaluation of these assumptions will uncover problems that redefine what has already been done, in effect, to change previous assumptions. This is in keeping with Malhotra's model in that an evaluation process involving the user reveals hidden or unstated requirements.

Also reflected in this model is the idea of design iteration. As with the Malhotra model, the design problem is attacked by developing sub-solutions and presenting them to the user, which in turn would modify the working set of assumptions. These iterations of assumption formation, design generation and design evaluation progress in an ever narrowing cycle until the point is reached where an acceptable solution is produced.

Summary of Design

In this section, we have explored models of the design process. It is significant that all three models portray design as an iterative process in which both the goals and solutions are developed in parallel using a feedback mechanism. Furthermore, we discussed the observation by Malhotra that consideration of provisional solutions uncovers new requirements. Later, these conclusions will be incorporated into a requirements exploration methodology.

Software Development Practices

The development of a software system is usually a lengthy and complex task involving the identification of a user's information needs, the designing of a system to meet those needs, and finally the practical application of the system. Accordingly, there exist various methodologies with which to manage software development projects and the communication within. In this section, we will look at current methods of software development.

After reviewing a method for selecting a requirements determination strategy, the traditional method of software development will be presented as the basis for subsequent discussions of rapid prototyping and the use of scenarios in determining user requirements. For each topic, a brief discussion of the method is presented along with a summary of its strengths and weaknesses.

Selecting a Requirements Determination Strategy

Davis [Da82] details a process for selecting a requirements determination strategy based on the the level of uncertainty associated with the problem environment. He defined a *strategy* as "an approach for achieving an objective" [Da82 p.12]. A *method* or *methodology* is the detailed means for achieving an objective.

To determine the level of uncertainty, Davis describes an evaluation used to assess four elements within the development process. Shown in Table 2-1, these elements are: the utilizing system, the information system, the users, and the analysts. From the evaluation of the elements, one of four strategies is then selected:

Asking is selected for the lowest level of uncertainty. It assumes that the users have a satisfactory way of structuring their information. Representative methodologies include the use of closed questions, open questions, brainstorming and the Delphi method.

Deriving from existing systems is associated with a somewhat higher degree of uncertainty. It involves using an implemented system with an operational history as a basis from which to elicit requirements. The focus here is on the data inputs and outputs of the existing system. Accordingly, this is applicable for fairly standardized operations.

Synthesis from characteristics of utilizing system is used in cases involving a high degree of uncertainty. In essence, it entails developing the requirements based on the activities of the object system. It is appropriate when a system is changing in its content, form, or complexity. Representative methodologies include; normative analysis, critical factors analysis, process analysis, and decision analysis.

Iterative discovery is used in the highest levels of uncertainty and is virtually synonymous with implementation prototyping. It involves capturing and implementing an initial set of requirements. This provides an anchor from which additional requirements are discovered and in turn implemented. This strategy is effective in cases where (1) there is no well-defined model of the information requirements, (2) the experience of the users and/or analyst is insufficient to define the requirements, or (3) the user's information needs are evolving.

To summarize, Davis's determination process is important to our discussion for two reasons. First, it provides a good overview of the strategies that can be employed to elicit requirements. But perhaps more importantly, it delineates the role of uncertainty in requirements exploration.

Table 2-1. Characteristics of elements in the development process [Da82 p.22]

Elements in development process	*Examples of characteristics that:* *Reduce uncertainty*	*Increase uncertainty*
Utilizing System	Stable, well-defined Programmed activities	Unstable, poorly understood Nonprogrammed activities
Information system or application system	Traditional, simple set of requirements	Complex or unusual set of requirements
Users	Few users, high experience	Many users, low experience
Analysts	Trained and experienced with similar system	Little prior training or experience with system

The Traditional Method of Software Development

To date, the most common and successful model of software development is what we will call the traditional approach. Also known as the waterfall model, it was defined in 1970 by W. W. Royce and refined in 1976 by Boehm. Developed during a time when software development was considered more of an art than a science, the waterfall approach applied an engineering problem-solving approach to software development.

Description of the traditional method

The traditional model is characterized by a step-by-step linear sequence of development phases from initial requirements gathering to the final system implementation. Although known by different names and variations, Sommerville [So85] describes the five phases of the traditional method as a) requirements analysis and definition, b) system and software design, c) implementation and unit testing, d) system testing, and e) operation and maintenance. Each phase produces a well defined output that becomes an input for the next phase of development (except for the operations and maintenance phase which feeds all of the previous phases). Control of the project is maintained via a series of baselines. Baselines are points at which the software system definition is formally reviewed, agreed upon by all parties (development and users), and then published as the new reference.

In the traditional scheme of things, requirements are developed solely during the requirements and definition phase. In this phase, a problem or opportunity is investigated and developed to produce the requirements specification. This document is then reviewed and agreed to by all parties, including the user. From this point, the requirements are assumed to be fixed, and software design begins. At project completion, the suitability of a system developed in the traditional style is judged by how well the resulting system meets the formal requirements specification.

Benefits of the traditional method

The traditional model has several benefits that can easily be understated. First and foremost, it provides a manageable framework emphasizing the management and communication aspects of systems development. As a result, progress on software development projects can be monitored and to some degree, controlled. This in turn reduces the risk associated with these projects, particularly for large, complex efforts. Also, as Boehm points out [Bo84], one has the ability to integrate many small programs into a large product using the traditional method. Finally, because the traditional model is the standard for developing software, it serves as a common reference throughout the industry.

Problems with the traditional method

Perhaps the main drawback to the traditional model is its rigidity. Feedback is limited primarily to reviews between development phases. This means that the users must know ahead of time, and be able to specify in detail, all of the decisions he is to make in order to know what information he requires [Ac67]. Accordingly, the requirements uncovered after the requirements development phase tend to be ignored until the maintenance phase when they are far more expensive to incorporate. Moreover, these post-implementation changes tend to deteriorate the functional structure within the software as well as undermining the relationships between the users and designers [St83].

A further problem concerns the form in which the traditional design takes. It is not until the implementation phase that code is written. Up to that point, the traditionally developed system is defined solely by documentation. Unfortunately, software is hard to visualize based on voluminous design documents. Brooks observed that "despite progress in restricting and simplifying software structures, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools" [Br87 p.12]. Consequently, there exists a huge and expensive gap between the time the requirements are established and the time their effects can be experienced by the users [Tu87]. As a result, by the time tangible results are available for the users to scrutinize, too much work has been done to allow for appropriate revisions [Sa87 p.7].

Additional problems with the traditional method, pointed out by McVay [Mc87], involve the emphasis of the development effort. He notes that traditional development methods are primarily *solution* rather than *problem* oriented. In practical terms, this means that by using the traditional scheme, we can be systematically solving the wrong problem. Moreover, McVay points out the lack of an explicit linkage between the user's problems and the solutions provided by the system.

Rapid Prototyping

In response to some of the aforementioned problems with the traditional method, a technique known as rapid prototyping has been gaining popularity. As does the traditional method, the concept of rapid prototyping had its roots in hardware development. However as Turner points out, the analogy between prototyping software and prototyping hardware is not a good one since hardware prototyping results in the first of a line of products, where software development is associated with a single product [Tu87].

Defining rapid prototyping

As the term prototyping has been used in varying contexts over the past several years, there is currently no universally accepted definition. The general intent is perhaps best captured by McVay who point out, "it may be safe to say that the purpose of prototyping is to provide pieces of a system to the user in the form of tangible, operating subsystems over time in order to get feedback sooner on how to proceed with subsequent implementations of the overall system" [Mc87 p.7]. James Johnson gives us a list of examples of prototyping [Jo83]. Among them:

- A fourth generation language working model that will be rewritten in a procedural language for implementation.
- A quick-and-dirty system intended to be enhanced over time until it is user acceptable.
- Mock-ups of reports. . . and screens. . .
- An experiment to decide if a proposed system is feasible.

As with any new technique, the users of prototyping have developed many different views of its use; hence the varied definitions. Accordingly, views of prototyping's relationship to the traditional model is also widely varied. Some writers have suggested prototyping as a replacement while others suggest it as a supplement to the traditional method.

Evolve or discard?

Perhaps at the heart of this issue is the disposition of the prototype itself. Some prototypes (described as throwaway) are built solely to describe the perceived behavior of a system to a user, and then discarded. This philsophy embodies Brooks' advice; "Plan to throw one away; you will anyhow" [Br75 p.72]. Accordingly, Gomaa argues [Go83] that prototypes are valuable as specification aids, but should not be released as a final product. He suggests the use of an interpretive language with powerful data manipulation features like APL to quickly develop a prototype emphasizing the user interface. Lehman also expressed this view in his perception of a prototype as "a validation model that displays sufficient characteristics of the desired system that, if satisfactory, could be used to fulfil the system role" [Le82]. The effect here would be to either supplement or replace the requirements specification phase of the traditional method.

Alternatively, a prototype could serve as a skeleton to be further developed into an end product, in effect replacing the traditional method. This view is reflected by the methodology developed by Mason and Carey [Ma82]. Their approach uses scenarios, prototyping, and a data-oriented design approach to approach applications that are highly interactive in nature. They argue that if an evolutionary approach is adopted, the tools used are different and geared towards maintainability. Additionally, they argue that in this evolutionary approach, there are no surprises to the user in the transition to the final version.

Prototyping classified by purpose

Providing some structure to the whole issue, Floyd categorized three approaches to prototyping based on the purpose of each [Fl83]. Moreover, these approaches are assessed with respect to their compatibility with the traditional method of software development. It should be pointed out, as Turner does, that these approaches are not mutually exclusive [Tu87]. Floyd's categories are as follows:

Exploratory Prototyping is done primarily to clarify the users needs. It is employed mainly to point out alternatives and to make the effects of a user's choice apparent. Exploratory prototyping is done in a throwaway manner and so is compatible with the traditional model.

Experimental Prototyping evaluates proposed solutions. The prototype is built to be evaluated in terms of feasibility, performance, or some other criteria. They can either evolve or be discarded. This approach is compatible with the traditional approach.

Evolutionary Prototyping is done primarily to cope with changing requirements. Here a system is viewed as a sequence of versions. The evolutionary approach implies a variance of the traditional model in that it is an iterative, miniature traditional life-cycle in each phase of the prototype's development.

Applicability of prototyping

Rapid prototyping is not appropriate in every situation. Davis prescribes general guidelines for determining system requirements [Da82]. His driving factor is the level of uncertainty associated with the application both on the part of the user and designer. Indeed, most of the literature dealing with prototyping suggests that it is best used in situations involving a user with incomplete or unclear requirements [Al84]. Related to uncertainty is prototyping's applicability in situations employing innovative technology or when approaches are used that may require a degree of experimentation [Al84].

Advantages of prototyping

The use of prototyping has a number of advantages over the traditional model with regard to the definition of user requirements. These advantages are realized in terms of communication, human relations and lower overall project costs. A more subtle advantage to prototyping is that it is geared more towards solving the right problem than simply fixing a set of requirements, then focusing on the solution.

The most important advantages are related to quickly communicating system capabilities to users in an understandable form [Al84, Go83]. Perhaps the biggest problem with the traditional approach is the length of time that users wait to see tangible results. Because they are developed as quickly as possible, prototypes provide a realistic view of how a system will satisfy users' needs much earlier in the process. The result is that changes, if needed, can be made much earlier when they are much cheaper to make.

As a means of communication, prototypes are easily understood. Using requirement specifications developed in the traditional method, it is very difficult for a user to see how his needs are being interpreted. Prototypes, on the other hand, are "real" and not subject to the levels of interpretation required using traditional requirements documentation.

As well as improving communication between designer and user, prototyping educates both. It is inevitable that designers learn something about the domain for which they are designing, and that users learn something restrictions they impose on the designer. As both parties assess the applicability of scenarios for given situations, they learn more about the needs and constraints of the other [Ta82]. Consequently, a more realistic product evolves from this learning process.

Other advantages involve the perceptions of the users. Since prototyping involves a higher level of involvement on the part of the user, the user has a more realistic view of the system and is more likely to accept the end product [Al84]. Illustrating the point, an industry survey of MIS managers conducted by Langle, Leitheiser and Naumann found a higher level of perceived satisfaction on the part of users and designers with prototyped systems [La84]. Another perception by the user is that of feasibility, or the ability of a design to meet the needs [Tu87]. As Boehm pointed out in his study [Bo84], it is easy to over-promise when specifying a system design. A prototype is concrete and real, and thus reassures the user that his needs can be met.

A final advantage, depending on one's perspective, is cost. Boehm and Gomaa argue that relatively high initial investment in developing prototypes is more than compensated for in terms of costs that would otherwise be incurred in correcting problems later in a project's life [Go83, Bo84].

Disadvantages of prototyping

There are several disadvantages associated with rapid prototyping but unfortunately, there has been relatively little research done on the use of prototyping. Research results presented here are based on surveys by Alavi [Al84] and Langle et al. [La84], and experiments by Boehm et al. [Bo84] and Alavi [Al84].

The major complaints with prototyping involve difficulties in managing and controlling prototyping efforts, especially in large, complicated projects [Al84]. This is mainly due to the lack of a well defined and accepted methodology associated with a lack of experience. One of the strengths identified with the traditional method was the use of well defined and accepted milestones and reviews to monitor progress. As of yet, no such control exists for prototyping. Two other issues related to management involve the tendency to ignore documentation and interfacing. Because prototyping emphasizes quick development, documentation is not a by-product of the process and interfacing issues are not adequately considered [La84, Sa87].

An additional disadvantage, depending on the perspective, is cost. Prototyping is a very expensive way to identify requirements when compared to requirements identification in the traditional method [Go83]. It is argued however that this initial expense is recovered in terms of costs that would otherwise be incurred correcting a traditional project late in its development or during its use [Go83, Bo84].

Finally, an increased involvement with users brings with it added problems in managing human relations. One such problem is the users' misconception of progress. After seeing a prototype running in a short period of time, the user may think the project is nearing completion and become frustrated as the details are worked out [Al84]. Also, users may lose interest after the initial prototypes have been reviewed and the details have to be considered.

Scenarios in Software Development

The main advantage of the prototyping approach to software development is more effective and timely communication between the users and designers of software. However, as noted by Hooper,

> The concept of prototyping is considered desireable, but in actual practice development costs have presented a major problem. . . . Prototyping clearly would be a very attractive requirements identification approach if low cost, ease-of-use, and timliness of results could be achieved. [Ho82 p.90]

To address this situation, it has been suggested that the advantages of rapid prototyping can be realized without the overhead of actually building prototypes by using *scenarios* [Ho82, Ma82, St83].

This section of the thesis explores scenarios and discusses the role they can take in determining user requirements. After defining scenarios and discussing their use, we will look at an existing methodology employing scenarios, and conclude the section with a discussion of the applicability of scenarios and their relative strengths and weaknesses.

Background

The idea of using scenarios as prototypes has appeared several times in computing literature. After defining scenarios, we will examine research that suggests the use of scenarios, or a similar concept, in the software development process.

Scenarios defined

Scenarios can be thought of as stories that illustrate how a percieved system will satisfy a user's needs. Thinking of scenarios as stories is relevant. As Crowley observed, "through stories we are able to have events brought before our minds. . . since they consitute a nearly universal form for the organization and dissemination of experience, they are an important means for creating social meaning and a shared sense of participation in both a common culture and a particular social order" [Cr82 p. 81]. Young and Barnard's definition is closer to the world of software development; they view a scenario as "an idealized but detailed description of a specific instance of a human-computer interaction" [Yo87 p. 291].

Hooper and Hsia

The term *scenario-based prototyping* was coined by Hooper and Hsia who suggested the use of scenarios rather than implemented prototypes to develop a user's requirements [Ho82]. They proposed the use of scenarios to capture the conceptual system as visualized by the users using simulation and man-machine interface techniques. They pointed out that while there are numerous methodologies for recording requirements and identifying inconsistencies, they have not really helped the users in identifying those requirements.

Scenarios, they proposed, could serve as the informal basis for communication in the role of prototypes. They said, "in prototyping by use of scenarios, one does not neccessarily model the system or any component thereof directly, but rather represents the performance of the system for a selected sequence of events" [Ho82 p.90]. They point out that in many cases, scenarios would obviously be quicker to formulate than full system models and allow for faster feedback on the part of the user.

They argue that to be effective, a method must be very informal enough to facilitate interaction with untrained users. The problem with the more formal systems is that they share the same major problems of implementation prototyping of cost and time delays due to their difficulty of use.

Hooper and Hsia's intention is that this research "be directed towards the development of a methodology for devising and using scenarios in requirements identification" [Ho82 p.90]. With respect to the lifecycle, they feel that once a set of requirements was achieved, the scenarios would continue to be useful as a guide to the continued development of the system through the testing and maintenance phases.

Steele and Nowell

"Prototyping at the conceptual level is much less costly than prototyping at the implementational level" [St83 p227]. Closely related to scenario-based rapid prototyping is Steele and Nowell's idea of *conceptual prototyping* [St83]. Their work is motivated by the recognition that "an unrecognized or unarticulated requirement is most often the impetus for costly post-implementation system changes" [St83 p.226]. Based on Malhotra's model of the design process [Ma80], they view the role of the designer as an idea generator rather than simply an interviewer or observer and acknowledge the role of provisional solutions in uncovering unstated requirements.

Steele and Nowell are not very specific about the process other than to say that conceptual prototyping involves presenting conceptual level sub-solutions, or conceptual prototypes, to a user for review rather than implemented prototypes [St83]. Certainly, scenario-based prototyping can be considered a form of conceptual protyping and the motivations for both serve to enhance the requirements exploration process in general.

Wexelblat

As part of MCC's investigation of the upstream process, Wexelblat [We87] has investigated the general application of scenarios in gathering behavioral data on a system. He views scenarios as an informal, natural way to describe how things behave and seeks to exploit this property, allowing for enhanced communication between users and designers.

He views specifications for an information system as being composed of two parts: functional and behavioral. Functional specifications are the more traditional statement of what a system is to do, involving interfaces, data, and processing. Behavioral specifications, on the other hand, describe "the way a system behaves, concentrating on interactions between system and users . . . from the point of view of an external observer or user" [We87 p. 2].

The idea of "gathering behavioral data" can be thought of as scenario-based prototyping; and although Wexelblat does not use the term, he hypothesized that scenarios and prototypes contain similar information. With respect to the life-cycle, he suggestes that scenarios could be used to determine which features of a system should be further developed using impementation prototyping.

Young and Bernard

In an interesting application of scenarios, Young and Bernard seek to apply them as early tests for proposed theories of Human-Computer Interaction (HCI) [Yo87]. Their idea is to use scenarios to quickly "weed out theories whose scope is too narrow for them to apply to many real HCI situations" [Yo87 p. 291]. Young and Bernard's interest is impelled by what they perceive as a dysfunctional preoccupation with accuracy associated with a cognitive science approach to developing HCI theories. This can be likened to the preoccupation with solutions in the traditional method of software development.[1] The suggested application for scenarios is very similar to those involved in defining user requirements.

It seems reasonable to liken a proposed software system to a proposed theory of HCI. Both involve building models of the user and both seek to balance accuracy and scope. Moreover, both HCI and software development run "the risk of failing to predict the behavioural consequences of a proposed design" [Yo87 p.292].

The process is somewhat different but the desired result, timely feedback, is the same. The difference being that the set of scenarios chosen for use in HCI are developed from empirical observations of the user, while scenario-based prototypes represent an

[1] A disadvantage to the traditional model discussed by McVay [Mc87].

imaginary perception of a user interfacing with a system. The objective is, of course, early feedback into the design process so that adjustments may be made, or theories discarded.

Using Scenarios

Having seen how several authors propose to employ scenarios in software development, we will in this section examine the use of scenarios in closer detail. First, based on Wexelblat's paper, the construction of scenarios is explored. Then we will examine an existing methodology employing scenarios and discuss the applicability of scenarios in general.

Constructing scenarios

Wexelblat explores scenarios and their use in detail [We87]. He discusses what information a scenario should contain, how it is constructed, and how it is organized. As mentioned earlier, a scenario tells a story. Essentially, it is nothing more than a description or a sequence of events used to explain something; instructional guides are a common example. For our purposes, a scenario represents a provisional solution to a user's problem.

Wexelblat describes a framework around which scenarios are constructed. With the scenario, he suggests including a body of background information including the title, author, date, version information, keywords and associated issues. Having this information grouped together makes reviewing the scenario much easier. Also, some structure is required if scenarios are to be maintained in groups. A benefit associated with the keywords is adoption of a common vocabulary among the parties involved and an accessible reference for outsiders. The issues involve capturing questions or implications that the scenario might raise such as advantages, disadvantages, impacts, or further actions.

When writing scenarios, an important feature of scenarios is flexibility; they can take on various forms such as text, pictures, or diagrams. Moreover, they can be structured in various ways such as dialogs or narrative descriptions. Accordingly, scenario authors should use the combination of form and structure that best suits their style of writing and best conveys their points.

To effectively communicate the desired information, caution must be used in their construction. To begin with, scenarios must contain the essential facts of the explanation, presented at the proper level of detail. Wexelblat warns of presenting "black boxes" to naive users in such a way as to solve a problem in a magical, unexplained way. Scenarios written at too high a level of detail may miss features that the users need to envision whereas too low of a level may force the user to sift through unimportant details. The objective is to capture the essence of the solution.

A scenario-based methodology

In at least one case, scenarios have been incorporated into a software development methodology. Mason and Carey describe a methodology employing screen-based scenarios as the first phase in an evolutionary prototype [Ma82]. The basis for this methodology is the architect's process where the system builder develops a view of the product based on its external description. Once the external view is established, the system is developed inward, maintaining consistency with the external view.

The scenarios employed here represent a screen-oriented dialog between the user and the system. This is accomplished using a predefined sequence of screens designed to behave like the final system. But because there is no application logic, the user follows a fixed script.

This methodology involves three iterative phases. The first of these involves the development of scenarios based on sequences of fixed screens. This phase ceases when agreement is reached on matters such as screen content and sequencing. The second phase pays particular attention to the details of data-dependent calculations where partial prototypes are developed to demonstrate actual database interactions and application computations on limited samples. The third and final stage involves the development of a prototype of the entire application which evolves to become the final system specification.

ACT1 is the tool that supports this methodology. It allows the development of individual screens simultaneously with the screen-linking logic associated with the data entries in those screens. In effect, it handles the creation and maintenance of the screens as well as the logic that controls their sequencing during scenario presentation.

Applicability of scenarios

How and when to use scenarios in requirements exploration are open questions which we will now explore. As we will discover, based on their generality and their associated cost of development, scenarios can perhaps be used most effectively in uncertain situations, possibly as a precursor to implementation prototyping. It has also been suggested that are best used to describe systems involving a high level of human interaction [Ma82].

Scenarios versus Prototyping. When a designer has formulated a design that he wants to communicate to a user, scenarios and prototypes both can provide the look and feel of the final system. Choosing between them strongly relates to Davis' method for selecting a requirements determination strategy based on a level of uncertainty [Da82]. In this case, there is a tradeoff associated with prototypes and scenarios in terms of cost and generality. The major factor in choosing between the two is the level uncertainty. The cost is the amount of time and effort required to communicate and generality relates to the ability to convey the system behavior in varying situations. The level of uncertainty is the risk that the design (or component thereof) communicated is not what was wanted.

Certainly, an implemented prototype can be applied to more situations than a set of scenarios. To see this, imagine of the number of scenarios required to thoroughly convey the behavior of even a simple implemented application. On the other hand, to convey a specific feature of a perceived system, an implemented prototype is very expensive when compared to developing a single scenario. The deciding factor is the level of uncertainty that what is communicated is what was wanted.

The level of uncertainty was described by Davis as being related to the experience of the participants and stability of the proposed system. Consider the following situations.

- A designer draws on an applicable existing design, so he needs only to demonstrate its use to the user. This corresponds to the *deriving from an existing system* strategy. With it, the cost is low because the system already exists. An existing system has the highest degree of generality, so the issue is that of uncertainty, how close will it be to what the user wants?
- A designer is very certain that the requirements are fixed, and that he has a well developed design model. He may simply develop the system in an evolutionary prototype. Here, the level of uncertainty is perceived to be low, the cost will be high, and the generality will be high.
- A designer is not so sure of the stability of the requirements, and/or the designer does not have a well formulated model (a higher degree of uncertainty), he may opt to describe specific features of the emerging design using scenarios. The generality of the scenarios will certainly be limited, but so will the cost.

So it seems that scenarios, because of their low cost and limited expressiveness, seem most appropriate for communicating specific system features in situations of high uncertainty. Wexelblat points out in his conclusions that the high level of conversation afforded by scenarios are most useful in cases where users only have a vague idea of what they need, and the system does not appear to duplicate existing ones [We87].

The type of system. Because scenarios capture the behavior of a system, they are probably best used to describe systems involving a high degree of human-computer interaction. Mason and Carey apply this characterization to Decision Support Systems (DSS) and Interactive Information Systems (ISS) as well as most business applications [Ma82]. However, meaningful scenarios could be constructed to convey more processing-oriented systems.

Advantages and Drawbacks of Scenarios

Having explored scenarios and their use in some depth, we are now prepared to describe their perceived advantages and drawbacks as a mode of communicating system behavior in requirements exploration. Unfortunately, there has been little empirical research on the use of scenarios in requirements exploration. Consequently, judging the merits of scenarios is largely intuitive.

Advantages

As was suggested in the introduction to this section, the primary advantages of scenarios involve their perceived ability to provide the early communication advantages of implemented prototypes at a cost far lower than that associated with developing implemented prototypes.

In terms of communication, scenarios have traditionally been a useful and accessible explanatory medium and a natural way for people to describe how things behave. Wexelblat describes scenarios as providing "a high-bandwidth communication path between users and designers, even when the two groups never meet" [We87 p. 2]. When scenarios are used, users are not forced to learn the jargon of the designer or forced to envision their system based solely on cryptic requirements documentation. Moreover, scenarios are a flexible communication tool in that they may take various forms such as narrative text, screens mock-ups, or slide shows: whatever effectively conveys the "look and feel" of the system.

Associated with this idea of accessibility is the informality associated with scenarios. This informality broadens the base of the users' contribution to the development process since there is no particular expertise required for the employment of scenarios [We87]. This alone could have a significant impact in that more of the actual end-users could become actively involved in the design process as opposed to being represented by a small group of designated representatives.

Furthermore, as with prototypes, scenarios can provide the focus for a valuable learning process for both designer and user. Boland explored the effects of a structured education process between users and designers and found that it generated more client-centered designs [Bo78]. Indeed, it is inevitable that the designer will have to learn something about the application world. Wexelblat likened a scenario to a window to the world of the customer: given enough windows, the designer can begin to anticipate the users needs [We87]. Likewise, users can be more easily made aware of the restrictions and tradeoffs associated with their needs.

Another potential benefit of scenarios is their usefulness throughout the development effort. Scenarios could be valuable references for the design, testing and maintenance of a system [We87, Ho82]. Additionally, scenarios could serve as training aids for newcomers to development or end users of the system.

Finally and perhaps most importantly, scenarios avoid much of the overhead in terms of time and effort of developing implemented prototypes. Wexelblat asserts that "it is clear that the effort required to converse via scenarios will be more than repaid by drastically lowering the cost of failed prototypes" [We87 p.10].

Drawbacks

As we have seen, scenarios lack the generality of an implemented prototype and are subject to a higher degree of interpretation on the part of the user. In levels of abstraction (i.e., the degrees of interpretation), implemented prototypes are, by definition, identical to the final system in critical respects, so subject to little interpretation. Scenarios on the other hand require a higher degree of interpretation than do implemented prototypes, but much less than traditional documentation. Additionally, scenarios must be developed to address a specific situation or problem and therefore do not lend themselves to experimentation in varying situations as do implemented prototypes. Another weakness of scenarios is due to their flexibility and informality which preclude formal mechanisms for dealing with completeness or inconsistency [We87].

An important additional concern is the perception of feasibility by the user. Scenarios may be viewed as contrived by a more cynical user. Reacting to the fulfillment of the promises made in an ambitious requirements specification, a subject in Boehm's study noted that "words are cheap" [Bo84 p.299]. Indeed, there is nothing in a scenario that assures feasibility as does an implemented prototype. Boehm noted that implementation prototyping gave the development team an earlier and more realistic view of what was feasible [Bo84]. Accordingly, Wexelblat warns that scenarios must must be written to avoid a black-box that solves a problem in some unexplained manner [We87].

A final problem is that of deciding what to represent in a scenario. As we have suggested, scenarios lack the generality of implemented prototypes; they apply only to specific situations and illustrate specific features. This places a burden on the scenario's author to accurately anticipate the important features and situations that must be represented[1].

[1]For a discussion of this issue, see [Ka88].

CHAPTER 3
A METHODLOGY FOR SCENARIO-BASED REQUIREMENTS EXPLORATION (SBRE)

In the previous chapter, we explored the cognitive processes and current practices involved in software development with respect to requirements exploration. This chapter proposes a general methodology for scenario-based requirements exploration (SBRE). The concepts and components of this methodology have been synthesized from the background study and are novel only in their integration into a requirements exploration methodology.

The chapter begins with an overview of objectives suggested by the material from the last chapter. With these objectives in mind, we then propose a high-level conceptual methodology for SBRE. To do so, a conceptual architecture of the information involved in SBRE is presented, then the mechanics of the methodology are discussed with respect to the architecture.

Objectives

As stated in the introduction, the objective of this research is to provide an effective method for conducting requirements exploration based on the use of scenarios as a means of communication between users and designers. Thus far, we have looked at the cognitive processes and current practices associated with developing software. This material suggests that to effectively accomplish requirements exploration, a methodology must have several features. Based on the discussion of the cognitive processes, the methodology should:

- Facilitate the parallel development of a high level design and a set of requirements regulated by a feedback mechanism [Am76, Ma80, Sa87].

- Provide for the evaluation of provisional solutions by the user in order to uncover unstated requirements or clarify misunderstood requirements [Ma80, Sa87].
- Provide a structure that would ensure that attention is paid to all aspects of the problem [Ma80].
- Employ an effective means of communication as a focus between users and designers that would overcome the barriers of communication to some degree.

Furthermore, based on the discussion of current software development practices, a methodology should:

- Appropriately emphasize the requirements definition process to ensure the correct problem is being solved [Mc87].
- Provide the user with early and usable feedback in a form that conveys the "look and feel" of the perceived system [Ho82].
- Provide for an explicit link between the system requirements and the solutions provided by the design [Mc87].
- Accomplish requirements exploration at a reasonable cost [Ho82].

With these objectives in mind, this chapter proposes a high-level conceptual methodology for SBRE. It should be understood that the methodology is not a replacement for the requirements and system design activities of software development, rather it seeks to coordinate them to yield a more realistic set of requirements.

Conceptual Architecture

Figure 3-1 shows the components of the conceptual architecture supporting SBRE. It is comprised of four sets of information that are captured and manipulated during the process. We call these the goal set, the scenario set, the design set, and the issue set. In our discussion, we will describe the information sets at a conceptual level to avoid issues of representation. Consequently, the formality of the design and goal sets depend on the techniques by which they are represented.

At any point during requirements exploration the architecture reflects the current information relating to the system in question. More specifically, the goal set represents the objectives of the system, the design set represents a plan (or plans) to meet the requirements, and the scenario set will link the goal set and design sets by showing how the design set achieves the goals. The issue set regulates the process by providing a mechanism for recording and organizing the issues that arise in the requirements exploration process.

As Figure 3-1 illustrates, the goal set is founded within the user's world and the design set exists within the designers world. The communication between these worlds is conducted via the scenario and issue sets. We will now discuss these sets in more detail.

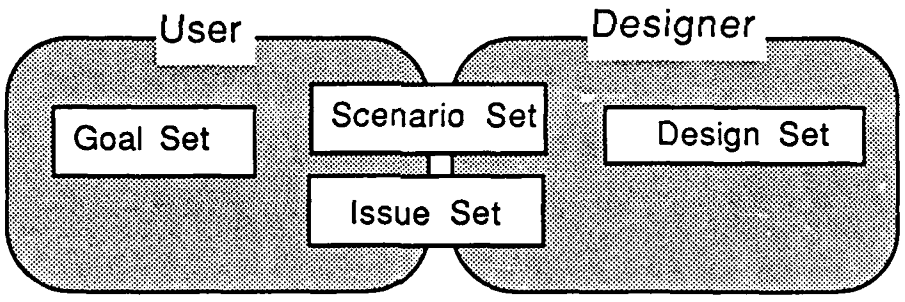


Figure 3-1. High level view of SBRE architecture

The Goal Set

Simply put, the goal set contains the current definition of what a system must do and within what means it must do it. This includes such information as requirements, constraints, standards, and available resources. Also included is relevant background material (i.e., general domain knowledge).

In the previously discussed models of design, the concept of the goal set is described in various ways. In the cybernetic model of design, the goal set corresponds to what Amkreutz calls the *input information* [Am76]. It is this information that determines the feasibility of the design and provides the criteria with which to order alternative design approaches. Sasso and McVay refer to this information as the *constraints* of the process which delineate feasible designs and against which possible solutions are tested [Sa87].

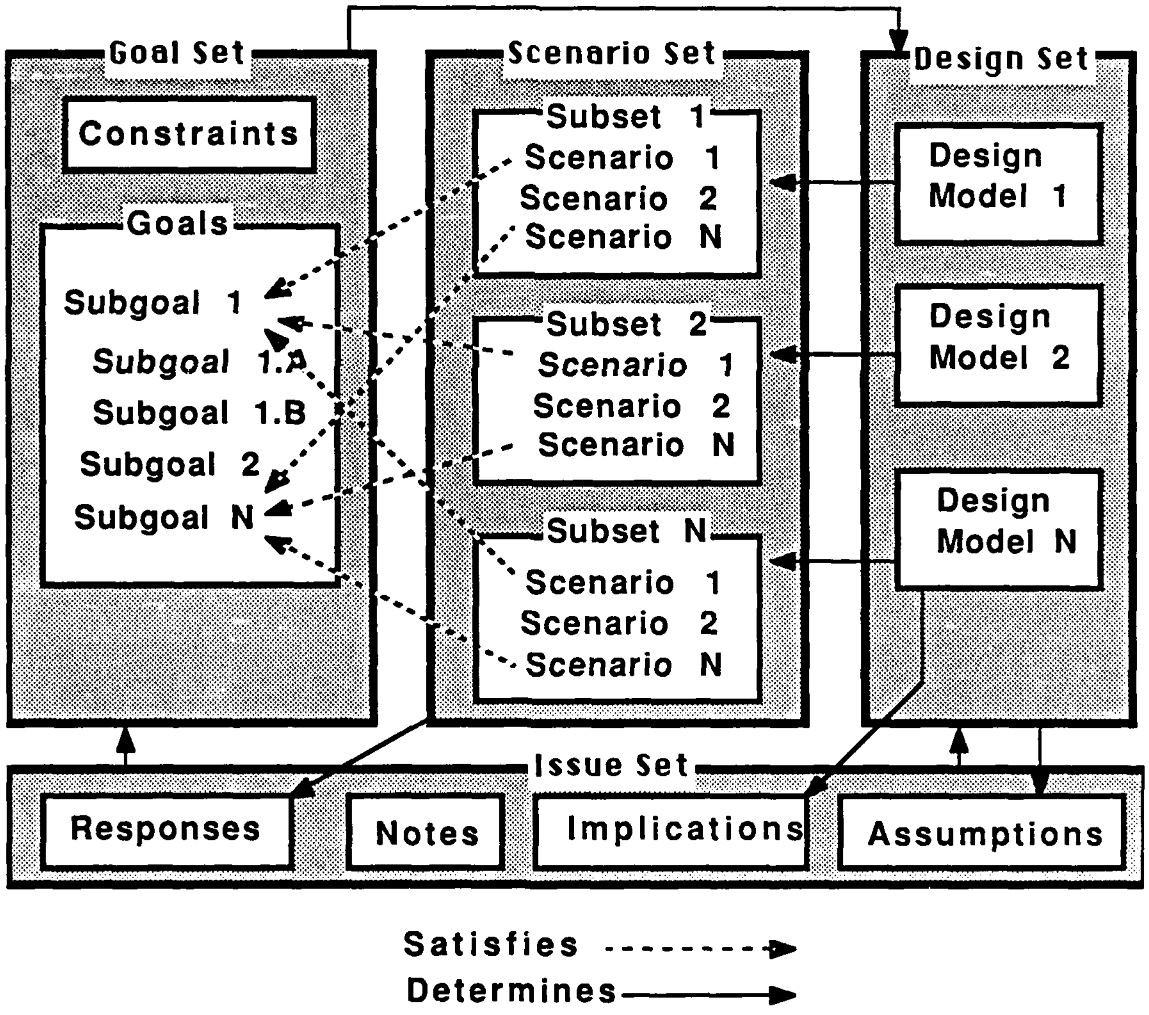


Figure 3-2. Detailed architecture

As illustrated in Figure 3-2, the goal set is represented as being a decomposition of the problem into a set of subgoals. Indeed, the first step for attacking ill-structured problems is to break them down into well-structured smaller ones [Si73]. Malhotra et al. described the decomposition process as continuing until the subgoals are specific enough to be considered as functional requirements [Ma80]. Moreover, Carroll et al. found that the presentation of problem information was a factor in how well it was solved [Ca80]. They found that a more hierarchially structured problem presentation results in solutions that reflect the structure of the problem and more stable design protocols.

As expected, the initial requirements can be be very fuzzy. However, as they evolve and become clearer, they should be recorded in such a way as to satisfy, as well as possible, the characteristics of a good Software Requirements Specification (SRS). The IEEE standards for requirements specifications [IE84] lists seven characteristics of a good SRS. A good SRS is: (1) unambiguous, (2) complete, (3) verifiable, (4) consistent, (5) modifiable, (6) traceable, and (7) usable during operations and maintenance. Additionally, the requirements should be prioritized, particularly in novel situations where the order in which the subgoals are attacked has a big impact on the ultimate solution [Si73, Gu87].

The Design Set

As described in the objectives, a requirements exploration methodology should support a parallel development of requirements and design. A design has been defined as a model, a plan for a system to be realized [Am76]. As provisional designs are formulated during requirements exploration, their distinctive attributes are recorded in the design set. While the form and detail of the design set may vary, at a minimum, it serves as a repository for the design decisions that will enable the construction of the envisioned system.

The design set corresponds roughly to what Conklin et al. call the *artifacts class* [Co85] and what Amkreutz calls the *output information* [Am76]. Both terms describe the tangible output of the design process consisting of both content and organization. The design set may also be compared to the *design space* of the Constraints and Assumptions model [Sa87]. The design space is the area in which solutions are considered feasible (not optimal) given the constraints recorded in the goal set.

Within the design set, there may be several alternative *design models* under consideration as shown in Figure 3-2. A design model should convey all of the relevant information that uniquely identifies a particular solution. These design models represent various resource configurations and/or software design variations and represent different approaches to achieving the goals.

The Scenario Set

The scenario set is the visualization of how the perceived system will accomplish the objectives defined within the goal set. The objective of the scenario set is to convey to a user what given features of a system would actually "be like" if implemented and capture the "look and feel" of how the requirements in the goal set will be satisfied by the perceived system. The scenario set serves as the focus for communication between the user and designer.

In essence, the scenario set could be viewed as what Wexelblat calls a *behavioral specification* in that it captures how the system reacts to its environment [We87]. But more than that, it can be used to convey the effective use of the perceived system within its operating environment.

As shown in Figure 3-2, scenarios are explicitly linked to the goals they satisfy as well as to the design model they represent. Moreover, because scenarios are dependent on the design model they represent, they are grouped into subsets that have a one to one relationship with the design models. A specific scenario can address many goals. As a result, the scenarios effectively link features of the solution or design to specific goals.

The Issue set

As the name implies, the issue set contains the issues that arise and are resolved during requirements exploration. The idea and need for the issue set was taken from MCC's work on the upstream activities [Co88, Co86, We87]. The motivation for an issue set is articulated by Wexelblat who notes: "the detailed design process that goes on in the dialogue between designers and customers can be viewed as the iterative process of discovering (or creating) issues and resolving them" [We87 p.11]. In effect, the issue set facilitates a feedback mechanism that supports the exchange and argument of viewpoints, ideas, concerns, etc. of the participants involved in the design process.

Specifically, an issue can be thought of a specific problem or question relevant to the project. Within the set are found four kinds of issues:

- *Assumptions* that are made by the designer in the face of perceived gaps in the existing set of requirements.
- *Responses* are the users' recorded reactions to a proposed solution as portrayed by a scenario.
- *Implications* are the constraints imposed on an overall solution and/or requirements based on the adoption of a partial solution. These may be thought of as "tradeoffs" associated with a particular solution component. Implications may be internal or external.

  External implications affect the requirements (usually involving a tradeoff in desired capabilities). Accordingly, they must be reviewed and understood by the user.

Internal implications act as constraints affecting future design considerations, they represent tradeoffs within the design model.

- *Notes* are general pieces of information or questions that may be relevant later in the requirements exploration process

These issues are linked to the specific aspect of the problem they address. For example, responses will be linked to scenarios, implications linked to design models, and assumptions linked to the appropriate level of the goal set. Notes can be associated with any part of the model.

Process

Now that we have seen what information is captured and how it is organized, the process of the methodology is presented using a data flow approach. It is based directly on Malhotra's [Ma80] model of the design process reflecting a cycle of goal elaboration, design generation, and design evaluation.

At a high level, the process employs a scenario generation/presentation cycle to take an initial set of requirements and iteratively refine it at the same time a high-level design is developed. More specifically, the design set, scenario set, and goal set are developed concurrently as regulated by the issue set. The result of the process is an equilibrium where the goal set accurately represents what the system is to do, the design set represents a feasible plan to meet the requirements. At this point, the scenario set will continue to serve as a focus for communication as well as a behavioral specification illustrating how the requirements will be met.

Initial Goal Elaboration

The first phase in the process is the initial goal elaboration. As depicted in Figure 3-3, this is where the initial set of system goals are formulated during the initial interactions with the user. At this point, information must be obtained in the more traditional ways such as interviewing or synthesis from existing methods [Da82]. In addition to the requirements and constraints, any relevant background information is captured such as profiles of typical users or existing methods. When the goals have been elaborated and decomposed to a point that the designer can begin to formulate an initial design model (or models), the iterative scenario generation/presentation cycle begins.

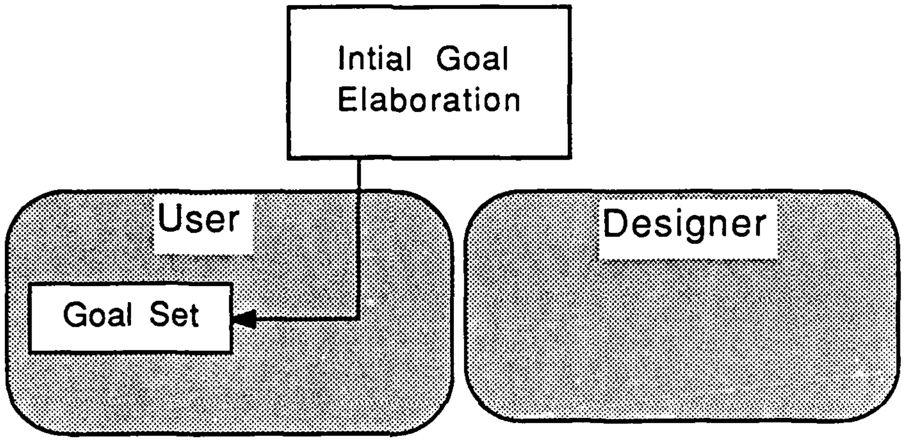


Figure 3-3. The initial goal elaboration phase

Scenario Generation Phase

The *scenario generation phase* is the design part of the cycle. This phase corresponds to the *generation function* in the cybernetic model of design [Am76] and Malhotra's *design generation* process [Ma80] in which a design is produced that approaches the current goals as closely as possible.

As illustrated in Figure 3-4, as the designer formulates design models of how goals can be met, the design considerations are recorded in the design set. Different design models are developed based on varying configurations of resources and/or varying processing algorithms. Based on the perceived behavior of these models, scenarios are then developed that demonstrate how the design model will satisfy specific requirements. With respect to the architecture, these scenarios complete a link between the design model and the specific requirement. In effect, this link shows how a design model will satisfy a particular goal.

Strategies for scenario development.

As was discussed in the previous chapter, scenarios are not as general as implemented prototypes or existing systems. Consequently, using scenarios to demonstrate all of the features or applications of an envisioned system is impractical. So the question becomes, what features are demonstrated? This question involves exploring the motivations for developing scenarios, which are based on the design strategy being employed.

As discussed earlier, an important characteristic of a problem environment is the level of uncertainty associated with it [Da82]. A high level of uncertainty is ascribed to situations in which neither the designer nor the user has a well-defined model of the requirements or solution. A situation such as this presents the designer with a *wicked* problem and must be solved accordingly. More specifically, unfamiliar or uncertain situations require a more *reactive* scenario generation strategy as opposed to familiar situations which permit a *enactive* strategy.

A familiar design situation is characterized by a designer who brings to it an applicable solution obtained either through experience or training. In a situation such as this, the emphasis is on the optimization or selection of an existing solution. Here, scenarios are formulated to allow (or force) the user to articulate essential system characteristics or select between various options. This assumes that there are no existing systems available to demonstrate, since it then would be easier to demonstrate the existing system rather than generate scenarios. For these situations, an *enactive* strategy is employed in that the impetus for the scenarios lies with the designer as he tries to constrain the design space. Software development testing strategies are a good way to ensure that the essential features of an existing model have been considered by the user [Ka88].

Alternatively, novel situations are those characterized by a high level of uncertainty and correspond to *wicked* or *ill-structured* problems. Accordingly, the designer does not have the experience or training to enter the problem with a well formulated design model. As Guindon observed, a more bottom-up approach is employed by designers to deal with unfamiliar situations [Gu87]. In such cases, the scenario generation process involves formulating a design model rather than simply bounding an existing one. The scenario strategy becomes *reactive* in the sense that the impetus for scenario generation lies with the user's existing goal set.

Two approaches to uncertain situations can be effectively employed as design strategies during scenario generation: the architect's process and serendipitous problem solving. As described earlier, a serendipitous approach involves solving a problem by moving between different levels of abstraction and detail and is driven primarily by the recognition of partial solutions [Gu87]. Because the goal set represents the problem in a hierarchial manner, the designer may first address the parts of the problem for which he is able to formulate a solution. As these subsolutions are formulated and validated by the user, they become the basis on which the rest of the problem is solved. The architect's approach is similar in that the more well-defined subgoals are addressed and the resulting interrelations of the subsolutions are incorporated into an overall design. Because merging the subsolutions can lead to inconsistencies, *internal implications* are used to record the constraints imposed on future design considerations by the adoption of a design component.

Issues generated during scenario generation

As noted in the discussion of the architecture, issues are problems or questions that arise during requirements exploration. In the scenario generation phase, notes, assumptions, and implications will be made in the following situations.

- In the absence of a perceived goal or constraint, the designer may make an *assumption* which represents a proposed *addition* to the set of requirements. This is recorded into the issue set and linked to the appropriate node of the requirements hierarchy.
- If the adoption of a design model (or feature therein) entails a *modification* to the existing requirements, an *external implication* is generated and linked to the appropriate goal. External implications can be best thought of as tradeoffs between functionality and feasibility.
- If the adoption of a design set component affects future design considerations, an *internal implication* is generated and linked to the affected design model within the design set. This is particularly important when employing a bottom-up approach to ensure consistency in merging subsolutions.

Scenario Evaluation Phase

Upon formulating a set of scenarios and their associated issues, the *scenario evaluation* phase of the cycle occurs. This phase acts as a feedback mechanism that reconciles the development of the designer's high-level design (i.e., the design set) and the user's requirements (i.e., the goal set). Scenario evaluation corresponds the *feedback function* in Amkreutz' [Am76] cybernetic model, and the *design evaluation phase* of Malhotra's [Ma80] model.

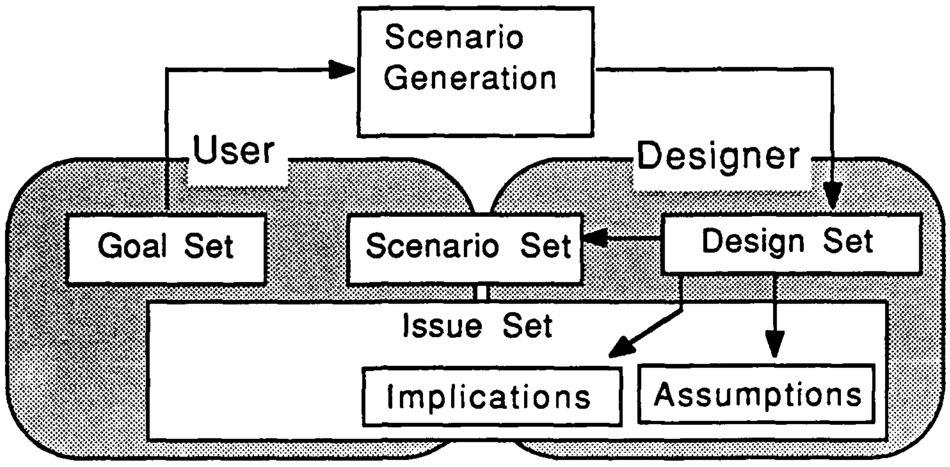


Figure 3-4. The scenario generation phase

In general, the scenario evaluation phase involves capturing the users responses to the design set as represented in the scenario set and modifying the goal set accordingly. This review of provisional solutions is the mechanism Malhotra found to uncover unstated requirements [Ma80]. Similarly, Conklin et al. note that these unstated requirements are traditionally called "mistakes" [Co85]. Complete scenario sets describing complete designs are not necessary prerequisites to scenario evaluation. The initiation of this phase is at the discretion of the designer and based on the criticality of the issues or scenarios to be reviewed.

As illustrated in Figure 3-5, the process entails a review by the user of a current set of scenarios along with any associated issues. The issues to be considered are any *external implications* and/or any *assumptions* that were generated by the designer during the scenario generation phase. The assumptions (i.e., perceived gaps in the requirements) are reviewed for their inclusion into the goal set. Similarly, the external implications are considered to review the effects on the goal set imposed by the adoption of a component within the design set. The effect of external implications on the goal set may be in terms of enhanced functionality afforded by the design, or alternatively, to restrict the goal set due to problems with feasibility.

As the user reviews the scenarios, his questions or disagreements are recorded into the issue set as ***responses***. A response may indicate that a scenario does not address features important to the user, in which case the scenario should be rewritten shifting its emphasis or detail. Alternatively, a scenario may be unacceptable, thereby indicating an unarticulated requirement or misunderstanding. In such a case, the user and designer together reevaluate the goal set to determine what changes are required.

Using the modified goal set, the scenario generation phase is again started. The cycle repeats until all subgoals have been addressed by user-approved scenarios that are accurate representations of the behavior of the design.

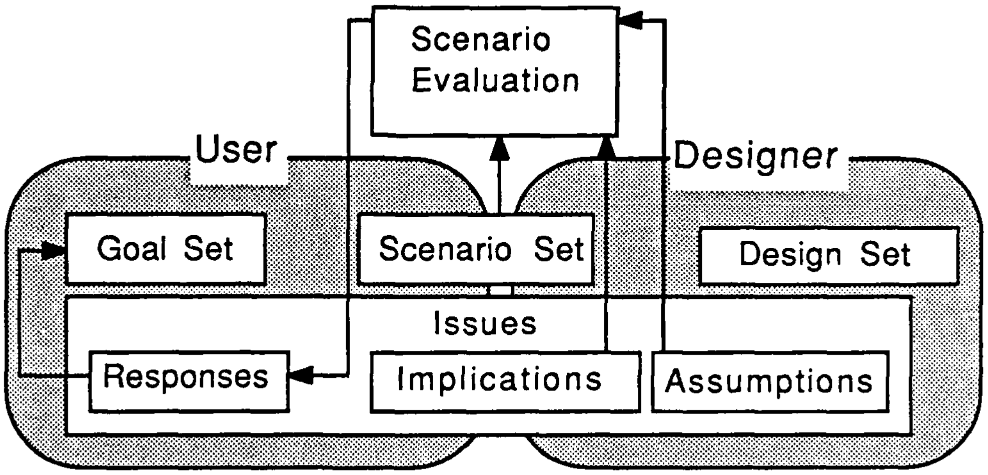


Figure 3-5. The scenario presentation phase

CHAPTER 4
TOWARDS AN SBRE TOOL

As we have seen, large-scale software design is a complex process of concurrent activities. Today's design tools, however, are focused primarily on one activity: the creation of artifacts that implement the design solution. Rarely are any of the other activities - problem formulation and solution identification and evaluation - supported. [Co85 p.3]

In the previous chapter an SBRE architecture and methodology was described. An important feature of the SBRE architecture is the linking between the information sets. Consequently, to support SBRE, a tool must aid in the maintenance of the information in these sets as well as establish and maintain the relationships between them. As it turns out, the power to do so is captured in a concept known as *hypertext*.

To support the methodology, this chapter describes the foundation for a hypertext-based tool that has been implemented using Apple's HyperCard™. We begin by looking at hypertext in general and then examine the concept and capabilities of Apple's HyperCard system. Finally, we consider how the SBRE architecture is constructed in HyperCard. Appendix B contains a more detailed description of the implementation.

Overview of Hypertext

In his survey of hypertext systems, Conklin [Co87] points out that most of today's computer systems process information in a linear fashion, hypertext systems on the other hand make extensive use of referential links to support *nonlinear* text. Two examples of what could be called manual hypertext systems are the use of reference books and note taking on 3x5 index cards. In both cases, embedded within the information they contain are references to other material, either to other note cards or other volumes.

Ted Nelson, one of the pioneers of hypertext, defined hypertext as "a combination of natural language text with the computer's capacity for interactive branching, or dynamic display. . . of a nonlinear text. . . which cannot be printed conveniently on a conventional page" [Co87 p.17]. Conklin explains: "the concept behind hypertext is quite simple. . . . Windows on the screen are associated with objects in a database, and links are provided between these objects, both graphically (as labelled tokens) and in the database (as pointers)" [Co87 p.17]. Figure 4-1 is an illustration of the relationship between windows and links in the display, and the nodes and links in the hypertext system. In this example, the link "b" has been activated in window A (using a pointing device such as a mouse). As a result, the new window B has been created on the screen, containing text from node B in the hypertext database.

Machine supported links are the essential feature of hypertext systems [Co87]. This capability makes hypertext very similar in some respects to semantic networks and relational databases. However, Conklin points out that these schemes "lack the single coherent interface to the database which is the hallmark of hypertext" [Co87 p.18]. To further illustrate the concept of hypertext, Conklin lists the characteristic features of a hypertext system:

- A database that is a network of textual or graphical nodes.
- Windows on the computer screen correspond to nodes in the database.
- Standard window operations are supported.
- Windows contain link icons which represent pointers to other nodes in the database. These link icons provide some indication of the contents of the node it points to.
- The easy creation of nodes and links.

- The database can be browsed in three ways: (1) following links, (2) searching the network for strings or attribute values, or (3) navigating the network using a graphical browser which provides visual clues to the content of a node.

According to Conklin, hypertext has to this point been applied in four general application areas: (1) *macro literary systems* which support large on-line libraries with interdocument links, (2) *problem exploration tools* which support the early unstructured thinking associated with problem solving and design, (3) *browsing systems* which are similar, but smaller in scale than macro literary systems, and (4) *general hypertext technology* which are general purpose systems allowing the application of hypertext to a range of applications such as writing, reading and collaboration.

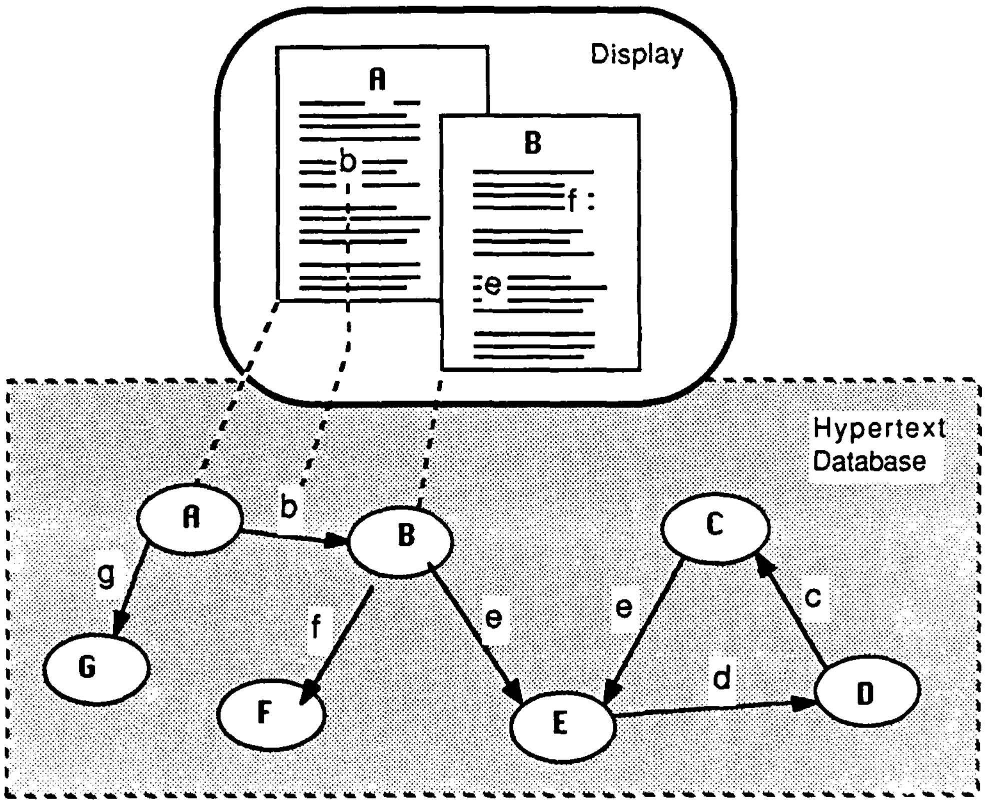


Figure 4-1. Hypertext relationships [Co87 p.18]

The problem exploration tools in general are very applicable to the SBRE methodology. These tools provide the required mechanisms for organizing, browsing, and filtering through a large collection of relatively unstructured information. More specifically, one subset of these tools, Issue-Based Information Systems (IBIS) support the human interactions required to approach *wicked* problems. The concept of an issue base was incorporated into the SBRE methodology.

Apple's HyperCard™

HyperCard is a hypertext system developed by Apple Computer, Inc. It is currently bundled with, and runs on, Apple Macintosh personal computers. Goodman notes that, "the [HyperCard] program is largely a metaphor for collections of information stored on cards, much like a card catalog at a library" [Go87 p.19]. HyperCard's author, Bill Atkinson, describes it as "an authoring tool and an information organizer. You can use it to create stacks of information to share with other people... it's both an authoring tool and sort of cassette player for information" [Go87 p.xxi].

The basic building blocks of HyperCard applications are *stacks, cards, fields and buttons*. The basic block, the *stack*, is usually a homogeneous collection of information similar to a drawer of a card catalog. The Macintosh operating system treats stacks as files. The HyperCard stacks are composed of *cards*. These cards represent the nodes of the hypertext database described previously. Using Goodman's analogy, a card in HyperCard corresponds to the card you would pull out in a card catalog drawer which typically contains one piece of information related to the content of the entire drawer. Unlike the windowing capabilities in Conklin's description, only one card is displayed at a time in HyperCard. Text associated with a card can be entered into *fields*, which usually contain the information that varies between cards. By using what is called a *scrolling* field, a card's boundaries can be extended for entering and displaying text.

Also located on a card are *buttons* which can be referential links to other cards or stacks. In this respect, they serve as navigational tools for traversing the HyperCard application. Although there is no real world analogy for buttons, if a card in a card catalog referred to a card in another drawer, using a button would automatically open the other drawer and find the card for you. Buttons are activated by clicking the mouse as the pointer points to the button's area of the card. As a result, the targeted card appears on the screen.

An important and powerful feature of HyperCard is its high-level processing language, HyperTalk. HyperTalk commands can be used interactively from the keyboard, or organized into scripts which are associated with any of the HyperCard objects. For example, the linking function performed by buttons is due to predefined HyperTalk scripts associated with the buttons. A script is activated by an event such as a mouse click that acts on the object with which it is associated. In addition to navigating through the application, HyperTalk commands can be used to create, delete and modify any of the HyperCard objects (stacks, cards, fields, or buttons) or their contents.

SBRE in HyperCard

Having discussed HyperCard's capabilities, we will now see how they can be applied to support the SBRE methodology. Figure 4-2 illustrates the SBRE architecture as it can be represented in HyperCard. In it, the SBRE information is represented in the five HyperCard stacks; (1) *goal decomposition*, (2) *constraints*, (3) *scenario subset*, (4) *design model*, and (5) *issues*. There could certainly be others, but these represent a minimum necessary to support the SBRE methodology. The arrows represent the links between the stacks and/or the cards. We will now discuss the sets individually.

The goal set.

As described in the last chapter, the SBRE goal set must minimally contain the system goals decomposed into subgoals to the point where they can be considered functional requirements. This goal hierarchy is maintained in the *goal decomposition* stack. To support this hierarchy in HyperCard, an arrangement of links as illustrated in Figure 4-3 would provide a simple hierarchic structure.

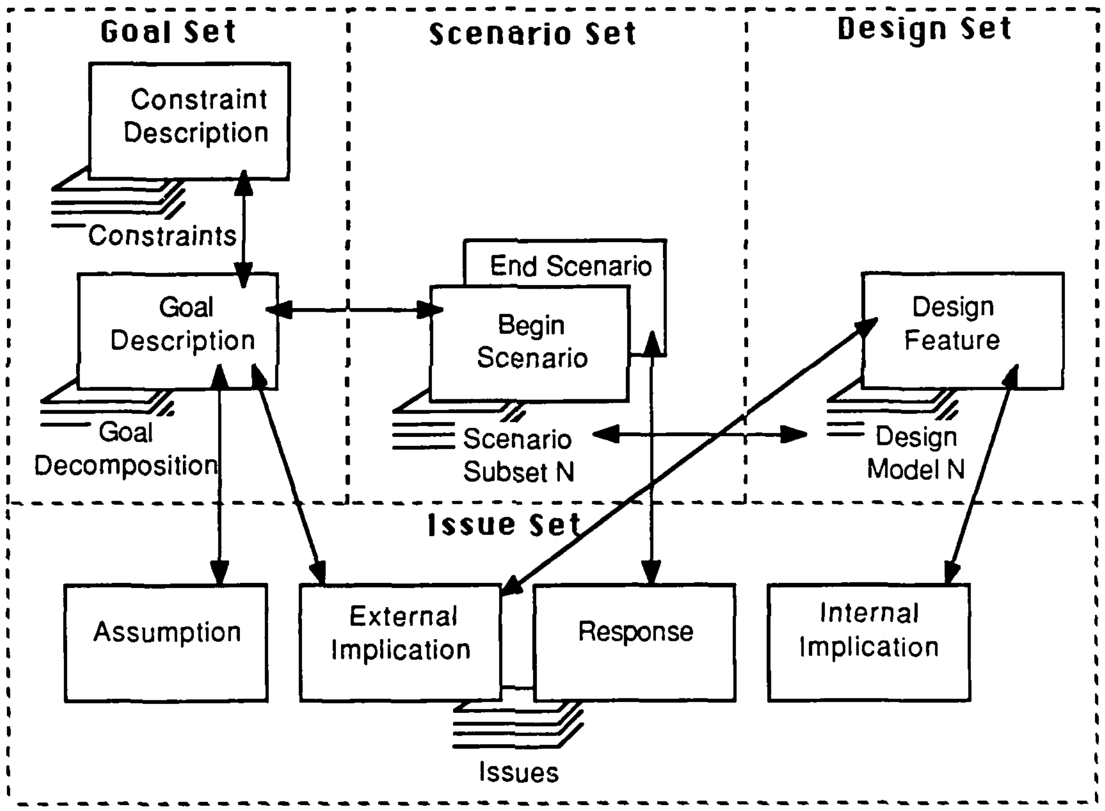


Figure 4-2. The SBRE Architecture in HyperCard

The goal decomposition stack is made up of *goal description* cards which contain three fields. One contains a short title describing the goal represented, a second contains the text description of the goal, and a third contains a list of the titles of any supporting goals. The supporting goals would be associated with a button linked to their description card as shown in Figure 4-3. Additionally, as illustrated in Figure 4-2, there may be buttons linking the goal description to constraints, scenarios or issues.

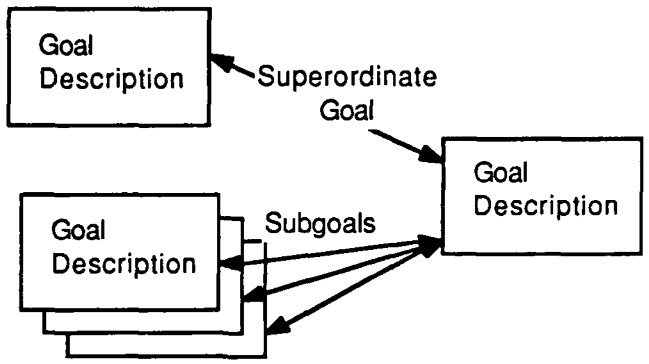


Figure 4-3. Hierarchy of Goal Description cards

Descriptions of the constraints would be maintained in the Constraints stack. A constraints description card would contain fields for a title and a complete description of the constraint. They are linked with the appropriate goal description card. This allows the constraints to be associated with the appropriate level of requirements. For example, the total time available for a project would be associated with the root goal description of the goal hierarchy. Other stacks could optionally be used to record other relevant domain information.

The design set

The design set must represent the design considerations for the different design models under consideration. Accordingly, each model would occupy a separate *design model* stack linked to its representative set of scenarios. The design set need only be visible to the designer, so the format of the stacks is subject to the design methodology used. For instance, in a data-oriented design scheme, individual cards could represent data entities that could be linked to represent their structure in a database. Additionally, cards representing different resources (i.e. people, hardware, or building space) configurations would be stored in the Design Model stack.

The scenario set

As shown in Figure 4-2, the scenario set consists of *scenario subset* stacks that have a one-to-one relationship with the Design Model stacks in the design set. Accordingly, all of the cards in the scenario stacks have buttons pointing to the design model stack. A simple way of constructing and storing scenarios would be as a sequential set of cards sandwiched between a *begin scenario* card and an *end scenario* card as shown in Figure 4-3.

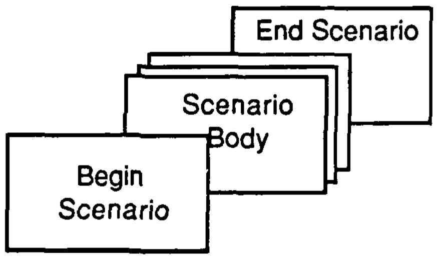


Figure 4-4. Scenario Organization

The begin scenario card would serve as an entry point for any links to the scenario. Additionally, this card could contain background information similar to that described by Wexelblat including the author, date, version and any keywords as well buttons to link the scenario with the *goal description* card(s) that it satisfies (illustrated in Figure 4-2.) The body of a scenario is simply a sequential set of one or more cards. Accordingly, they are traversed in sequential order from the begin card. Isolating the scenario body from the begin and end cards allows scenarios to be created without the overhead associated with maintaining the SBRE process. The *end scenario* card would serve to delineate the end of the scenario and contain the buttons with which to establish *response* issues. This above representation would work for completely independent scenarios. It certainly is possible however to interrelate scenarios in some sort of hierarchy of detail. Such a scheme is a natural application of a hypertext system.

The issue set

The issue set as illustrated in Fig 4-2 consists of a different type of card for each type of issue (this could be done simply using different card names.) The cards all have a title field as well as a scrolling field for the textual description of the issue. Also included is the button linking the issue to the description card containing the information in question.

It has been said that a record of the design decisions and their rationale would be useful in that it would provide a mechanism to backtrack and "unmask" the cause of mistakes [Co85]. The issue set as implemented in HyperCard could optionally satisfy this function through the use of an *archive* stack. This stack would simply be a repository of resolved issue description cards. As an issue is resolved, the card is transferred from the Issues stack to the archive stack and annotated with the date and a description of the disposition of the issue. In so doing, a history of the SBRE process would be maintained.

Functionality

Having detailed how an SBRE architecture could be formulated using HyperCard, we are prepared to discuss the high-level functionality of a SBRE tool in HyperCard. At a minimum, such a system should initially create, and support the expansion of the SBRE information sets. Additionally, to supplement the limited browsing capabilities of HyperCard, the tool must provide indexing capabilities to easily access the stacks.

Implementing an SBRE architecture

The primary function of an SBRE tool would be to create and maintain the SBRE information sets. The it must provide the framework into which the information will be entered by the designer and user. Initially, this will require that the goal decomposition stack be created with a root goal description card. Also, an empty design model stack would be established. As the design and scenario sets are generated, the tool must generate the new cards and maintain the links between them. Because issues will be formulated outside of the issues stack, a mechanism should be provided so that an issue card can be created from within any other stack and automatically be linked to the card from which it was created.

Browsing the SBRE stacks

In the general introduction to hypertext was described a browsing capability using icons that represented the nodes in the hypertext database. While HyperCard provides the means for sequential browsing and reviewing recent cards, to efficiently browse HyperCard stacks, additional facilities must be built into the tool to provide indexes and overviews of the information contained in the system. This can be done by building separate system index cards that would build a list of card titles and buttons linked with their description cards. To do so, the index card would contain a HyperTalk script that would make a pass through the stack and compile a list all of the card titles, location, and the presence of links.

Creating Scenarios

Using the structure suggested in the previous section, the body of a scenario is simply a sequential set of one or more cards used to illustrate an aspect of the proposed system's behavior. HyperCard offers a great deal of power for creating scenarios, enabling a range of sophistication from simple textual narratives stored in fields to full featured simulations involving cards that represent CRT displays, fields that accept and process user input, and HyperTalk scripts to mimic the system's behavior.

Summary

This chapter described the considerations for employing a hypertext-based tool to support the SBRE methodology. We began by discussing the concept of hypertext and how it applied to the SBRE architecture. We then examined the capabilities and features of Apple's hypertext system, HyperCard, and finally, described how the SBRE architecture can be represented in such a system. Appendix B details an implemented prototype of such a tool, and in the next chapter, we will describe some initial reactions to its use.

CHAPTER 5
SUMMARY AND CONCLUSIONS

Conclusions -- Assessment of Objectives

In the first section of Chapter 3, we outlined several features that should be incorporated into a requirements exploration methodology. We will now review these and describe how they are reflected in the SBRE methodology.

- Support the parallel development of both a high level design and a realistic set of requirements regulated by a feedback mechanism. The inclusion of both the design and goal sets in the SBRE architecture provides the framework to support these processes in parallel. The scenario evaluation phase in the SBRE process and the issue set in the architecture together provide a feedback mechanism regulating the design and requirements development processes.
- Promote the evaluation of provisional solutions by the user in order to uncover unstated requirements or clarify misunderstood requirements. Malhotra found that discussions between users and designers about how designs meet goals uncover new or unstated goals. This is the motivating force behind the scenario evaluation phase where scenarios representing the design are evaluated by the user so that the goal set may be adjusted accordingly.
- Provide a structure that would ensure that attention is paid to all aspects of the problem. The first step in dealing with complex problems is to decompose them into smaller and more manageable subproblems. However, there is a tendency to neglect subproblems when formulating a design. Providing explicit links to the goals as they are satisfied by scenarios will identify goals

that have not been addressed. A supporting tool should highlight such situations.

- Employ a means of communication between users and designers that would, to some degree, prevent the induction of noise into the communication process. This objective is satisfied by using scenarios as a means of communication. Since scenarios are flexible, they may be written at a level of detail that captures the essence of the solution without overloading the users with detail. Furthermore, design jargon is reduced because scenarios are written in the language of the user.
- Provide the user with early and usable feedback that provides the "look and feel" of the perceived system. This objective is achieved by the use of scenarios, which we have shown to be a fast and powerful means for communicating a system's behavior. Additionally, the process ensures that feedback is provided to the user before any resources are spent on an implementation.
- Provide for an explicit link between the requirements and the solutions provided by the design [Mc87]. The linking mechanism in the SBRE architecture explicitly associates design functions to specific requirements through the scenarios.
- Accomplish requirements exploration at a reasonable cost [Ho82]. Currently, implementation prototyping is the most effective way for a user to evaluate the suitability of a design. Unfortunately, this is a costly way to perform requirements exploration, especially in situations involving a high degree of uncertainty. Hooper rightly argues that the cost of developing scenarios is very low when compared to the cost of failed or inappropriate prototypes [Ho82].

To summarize, the SBRE methodology should prove to be a fast and effective way to elicit requirements. There are however some open issues associated with this approach which we will now discuss.

Future Research and Open Issues

Scenario selection

Throughout this thesis, we have taken a rather simplistic view of representing a system's behavior with scenarios. An important issue concerns identifying which facets of a design to represent and to what level of detail.

To effectively represent the design's behavior, the scenarios used must adequately "cover" the requirements in the goal set. It has been found that many of the coverage strategies used in software testing are directly applicable to requirements exploration [Ka88]. Particularly applicable are black-box testing strategies, which are based solely on the specified requirements. Two examples of black-box testing strategies are *equivalence partitioning* and *cause-effect graphing*.

Equivalence partitioning involves partitioning the input space of a program in such a way that an element of a given partition will be handled by the program in exactly the same way as any other element in that partition. With respect to scenarios, overall coverage is achieved when each partition of each input is covered by at least one scenario.

Cause-effect graphing differs from equivalence partitioning mainly in the level of coverage provided. Coverage with cause-effect graphing is achieved when every valid combination of input partitions, or causes, is covered. A *cause* is essentially an input partition while an *effect* is some system-level outcome. The requirements are used to develop a boolean graph which reflects the logical relationships between the causes and effects. By starting at a given effect and working backwards through the graph, every combination of causes leading to that effect can be identified and covered by a separate scenario.

Strategies may also be employed for purposes other than simply "covering" the design. It has been found that many of the knowledge acquisition strategies utilized in expert system development are directly applicable to requirements exploration [Ka88]. These techniques may be used to develop scenarios that selectively focus a user's attention on specific questions or problems.

Additionally, scenarios can be developed to demonstrate the *best use* of a system within the user's environment. As we have said, an implemented prototype can be very general in that it can be applied as the user sees fit. Unfortunately, the user may not have the insight required to best apply a system. Scenarios offer a way of illustrating the system's functionality and its optimal use.

As noted earlier in the thesis, scenarios illustrate specific features of a system for specific situations. As a result, it is infeasible to try to cover *all* features of a system for *all* possible situations. Therefore, the scenario set should be developed to convey the behavior of a design which is of interest to the user, while at the same time making the user aware of any inconsistencies in the requirements.

Relationship of SBRE to software development methodologies

Another open issue involves the disposition of the information contained in the architecture after the SBRE process. To a large degree, this will be determined by the level of detail of the information. If the SBRE process is conducted rigorously, and the uncertainty associated with the goal set is minimal, the SBRE methodology can serve as a precursor to the traditional software development methodology. In this case, the goal set would serve as a requirements specification, and the design set would serve as a design specification. Alternatively, if the process is conducted less rigorously, or there is still significant uncertainty associated with the requirements, the SBRE methodology could serve as a precursor to prototyping. Also, as suggested by Wexelblat [We87] and Hooper [Ho82], the scenario set may be used later in the development process as a guiding framework for implementation and testing of the system.

Development of a supporting tool

A prototype tool (described in Appendix B) has been developed to support the SBRE methodology and is currently undergoing an initial shakedown study. It is being used to explore the requirements for a process-oriented application involving a high degree of uncertainty. The participants are three graduate students, two serving as designers, and one as a user. The initial findings have been encouraging. The goal set changed considerably based on the reactions to scenarios and the issues generated. Not surprisingly, the designers felt that constructing scenarios forced them to focus more closely on the user's requirements and be more thorough in the design process than would have been the case otherwise.

Gaining a body of experience

Devising a way to evaluate the effectiveness of the SBRE methodology will be very difficult. How can one measure the ability of a particular methodology to uncover requirements? Perhaps the best way to evaluate the methodology, as well as address the issues above, is to develop a body of experience by applying it to a variety of realistic problems. In so doing, the following issues must be addressed:

With respect to the overall methodology:

- Define the types of problems for which the SBRE methodology is most effective. This can be characterized in terms of uncertainty and problem domain.

- Assess the acceptance of the methodology by users. Does the SBRE process have to be "sold"? Do users really understand what is being expressed in scenarios? Do the users tire of the process?
- Identify problems in managing the SBRE process. Determine the cost effectiveness of using SBRE. Develop guidelines for scenario selection criteria. Determine the compatibility of the SBRE process with established software development methodologies.
- Explore the compatibility issues involved with using various specification representations.

With respect to the tool:

- Explore the possibility of providing scenarios to users to be reviewed in the user's environment without designer assistance.
- Investigate provisions to support multiple users.
- Evaluate the suitability of the tool for various specification representations.

Summary

In Chapter 1 we established a need for research in, and improvements to, the requirements exploration process. This was done by identifying a causal link between inadequate requirements and what is known as the *software problem*. Furthermore, we described our intent to develop an effective method for requirements exploration based on the use of scenarios as a basis for communication between user and designer.

In Chapter 2, we presented related background material. We explored the cognitive processes involved in requirements exploration including communication, problem solving and design. Additionally, methods of software development were surveyed and a means for selecting a requirements determination strategy was presented. Finally, we examined scenarios as they apply to requirements specification.

From this chapter, some important conclusions were drawn. First, design problems are usually *wicked* problems, the solutions of which must be judged by criteria developed in parallel with the solutions [Co85]. Second, the design process involves the iterative and concurrent development of requirements and design [Ma80, Am76, Sa87]. Third, exploring solutions to problems uncovers unstated requirements [Ma80]. Fourth, scenarios offer a fast, effective, and inexpensive way to describe the behavior of a hypothetical system [Ho82, We87]. Fifth, it is important to provide users with early and useful feedback on how their requirements have been interpreted and how they are being met.

Based on the above observations, Chapter 3 proposed the architecture and process of an SBRE methodology. The first section described a conceptual architecture consisting of four sets of information captured and manipulated during requirements exploration: the goal set, the design set, the scenario set, and the issue set. The second section described the SBRE process as the formulation and refinement of these information sets using an iterative cycle of *scenario generation* and *evaluation*. The process stops when no open issues remain in the issues set. At that point, the goal set represents what must be done, the design set describes the plan for doing what must be done, and the scenario set describes how the system will behave when implemented. Appendix A provides a hypothetical example of SBRE use.

Chapter 4 examined considerations for developing a tool to support the SBRE process. Also, the features of a commercial hypertext system, HyperCard™, were outlined. Based on its structure, we then described how the SBRE architecture could be represented. Appendix B provides a more detailed description of a prototype tool based on this platform.

Finally, in this Chapter we have concluded that the SBRE methodology should provide an effective way to explore requirements. However, we also have identified the need to develop a body of experience to validate this conclusion.

APPENDIX A
THE LIBRARY SYSTEM -- A SCENARIO ILLUSTRATING SBRE

The "small library database" is a common demonstration problem for specification work [We87]. Here, the mechanics of the SBRE methodology are illustrated using a hypothetical library problem. The example encompasses a single iteration of the scenario generation/evaluation cycle.

Initial goal formulation

During the initial goal elaboration phase, through a set of interviews with the user, the following initial set of requirements and background material was obtained and recorded into the goal set .

System description -- School Library System

Hierarchy of tasks:

1. Maintain Loans
   - a Check out
   - b Return books
2. On-line card catalogues
   - a Query by author, subject, or title
   - b Indicate availability or due date
3. Maintain book inventory

Constraints:

1. All copies in the library either available or checked out.

2. No books may be loaned to borrowers with delinquent loans.

System users and privileges

1. library staff - access to all functions
2. library assistants - access to check-out and return functions
3. students - access to on-line card catalogue

Data currently maintained manually:

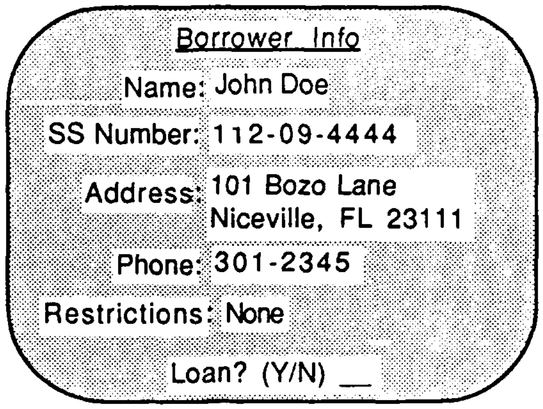
1. user information - name, address, phone number, social security number.
2. book information - call number, title, author, type of publication, date, publisher.
3. overdue list - overdue loans listed by borrower, title and date due

Scenario generation

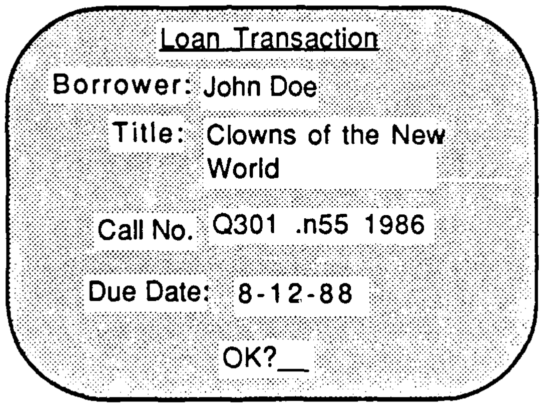
Based on the current information captured in the goal set, the designer formulates two design models. Both would use an on-line database comprised of the information now contained in the manual system. The difference is that one model, in addition to CRTs, would use scanners to read barcodes encoded on the volumes and library cards. These configurations of hardware are captured along with the database concept in the design set. To illustrate how requirement 1.a (Check out books) is to be satisfied by the two design models, the designer develops the following scenarios:

BEGIN SCENARIO: Checkout Using Scanner

- A borrower arrives at the assistance desk with some books to be checked out.
- The library assistant asks the borrower for a picture ID and his library card containing the user's barcode.
- The assistant scans the card with the scanner and the following screen appears on the terminal:


Borrower Info
Name: John Doe
SS Number: 112-09-4444
Address: 101 Bozo Lane
Niceville, FL 23111
Phone: 301-2345
Restrictions: None
Loan? (Y/N) __

• The assistant checks the screen to see if there are any restrictions to the borrower's privileges. If not, the assistant enters a "Y" into the "Loan?" prompt, and scans the book's barcode, producing the following screen:


Loan Transaction
Borrower: John Doe
Title: Clowns of the New World
Call No. Q301 .n55 1986
Due Date: 8-12-88
OK?__

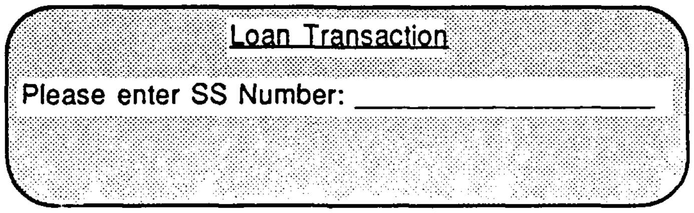
• The assistant checks the information, stamps the book with the due date, and enters "Y" at the "OK?" prompt. At this point, the system records the loan and updates the information on the availability of the book.

END SCENARIO

BEGIN SCENARIO: Checkout Without Scanner

• A borrower arrives at the assistance desk with a book to be checked out.

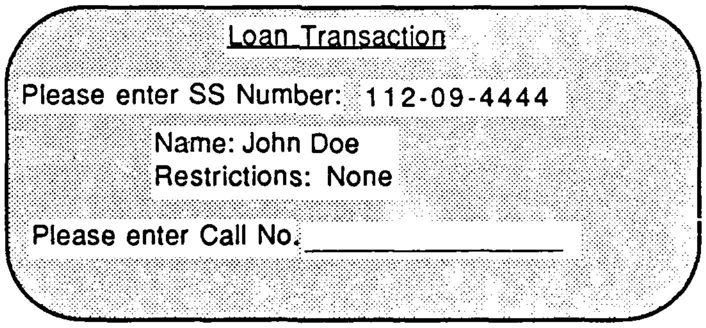
- The library assistant asks the borrower for his/her student id which contains the borrower's social security number.
- The assistant enters the social security number into the following screen:



Loan Transaction

Please enter SS Number: ___________________

- The assistant checks the response to see if the borrower's privileges are restricted for any reason. If not, the book's call number is entered on the screen:



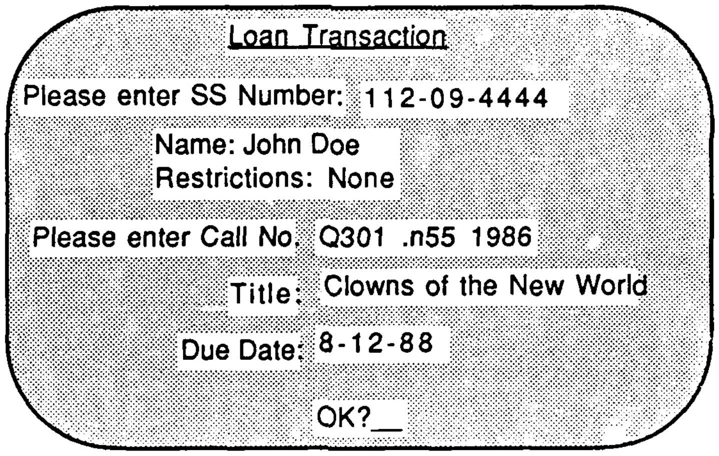
Loan Transaction

Please enter SS Number: 112-09-4444

Name: John Doe
Restrictions: None

Please enter Call No. _________________

- After the call number is entered, the book's title and the due date for the loan are displayed on the screen:



Loan Transaction

Please enter SS Number: 112-09-4444

Name: John Doe
Restrictions: None

Please enter Call No. Q301 .n55 1986

Title: Clowns of the New World

Due Date: 8-12-88

OK?__

• The assistant enters a "Y" at the "OK?" prompt and at that point, the volume is on loan to the borrower.

END SCENARIO

As he ponders the design and generates the scenarios, the designer realizes that the current manual library system maintains information for the borrower, but no provision for doing so exists in the current requirements. Consequently, he generates an *assumption* stating that a requirement needs to be added to this effect.

Also he realizes that the use of a scanner would require borrowers to have special barcoded library cards and all of the books to be labeled with barcodes. This is recorded in the issue set as an *external implication* since it entails a change to the existing goal set driven by the adoption of a component of a design model.

Scenario evaluation

After generating the scenarios and recording the issues, the designer decides that before going any further, it would be a good idea to see if the scanner-based design model is worth pursuing from the user's point of view. *Going into the scenario evaluation* phase, the SBRE architecture is as shown in Figure A-1. There are two design models differing only in that one model includes a scanner and the other does not. Based on these models, there are two subsets of scenarios, each containing a scenario illustrating the check-out procedure. Also there exist two issues for review, the assumption that information on the users must be maintained, and the external implication of adding barcodes to the books to support the use of scanners.

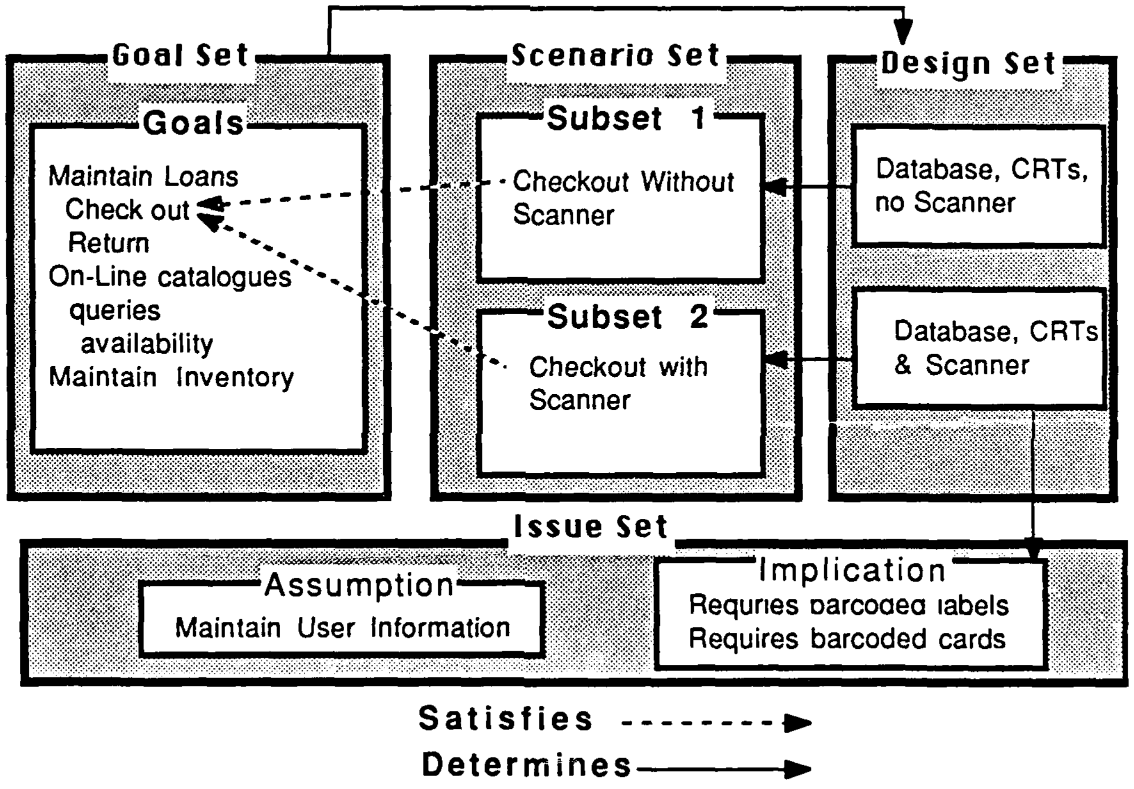


Figure A-1. Architecture entering scenario evaluation phase.

During the scenario evaluation phase, the designer presents the assumption concerning the maintenance of user information. The user acknowledges this omission explaining that currently, the student's ID card, which is issued by the school, is used.

Accordingly, a new requirement is added providing for the maintenance of user information within the library, and the assumption is removed as an open issue.

At this point, the two scenarios are reviewed. The first scenario, "Checkout Using Scanner", is reviewed along with its associated implications. The user is impressed with the speed at which checkout can be done with the scanner. However, as he reviews the associated implications, he realizes that using the scanner entails adding barcodes to all of the volumes in the library, as well as issuing library cards, which was something he hoped could be avoided.

After the second scenario, "Checkout Without Scanner" is reviewed, the user says he realizes that the scanner version would save his staff a lot of time and would speed up the checkout process considerably. However, he does want to know how a due date is being calculated and how would the system handle holidays and weekends. His reactions and questions are recorded as *responses* in the issue set.

After the responses are recorded, the user and designer together review the goal set with respect to the responses. They modify the goal set to accommodate the use of the scanners by adding the requirement that the books will be affixed with barcodes and the students will be issued barcoded library cards. During the discussion of the due date, the designer explains that his design calls for a fixed period of time to be added to the current date and adjusted for holidays using a standard calendar routine. The user responds that seniors can check out books for longer than underclassmen. The designer points out that there is no status or class field and that this needs to be added to the user information. As a result, a status field is added to the student information.

As a result of this cycle of the scenario generation and evaluation phase, the goals have changed significantly. The resulting architecture is shown in Figure A-2. From this point, the designer now has a revised goal set with which to begin another iteration of the scenario generation phase. The cycle would repeat until the user is satisfied that the scenarios represent an acceptable system for his needs.

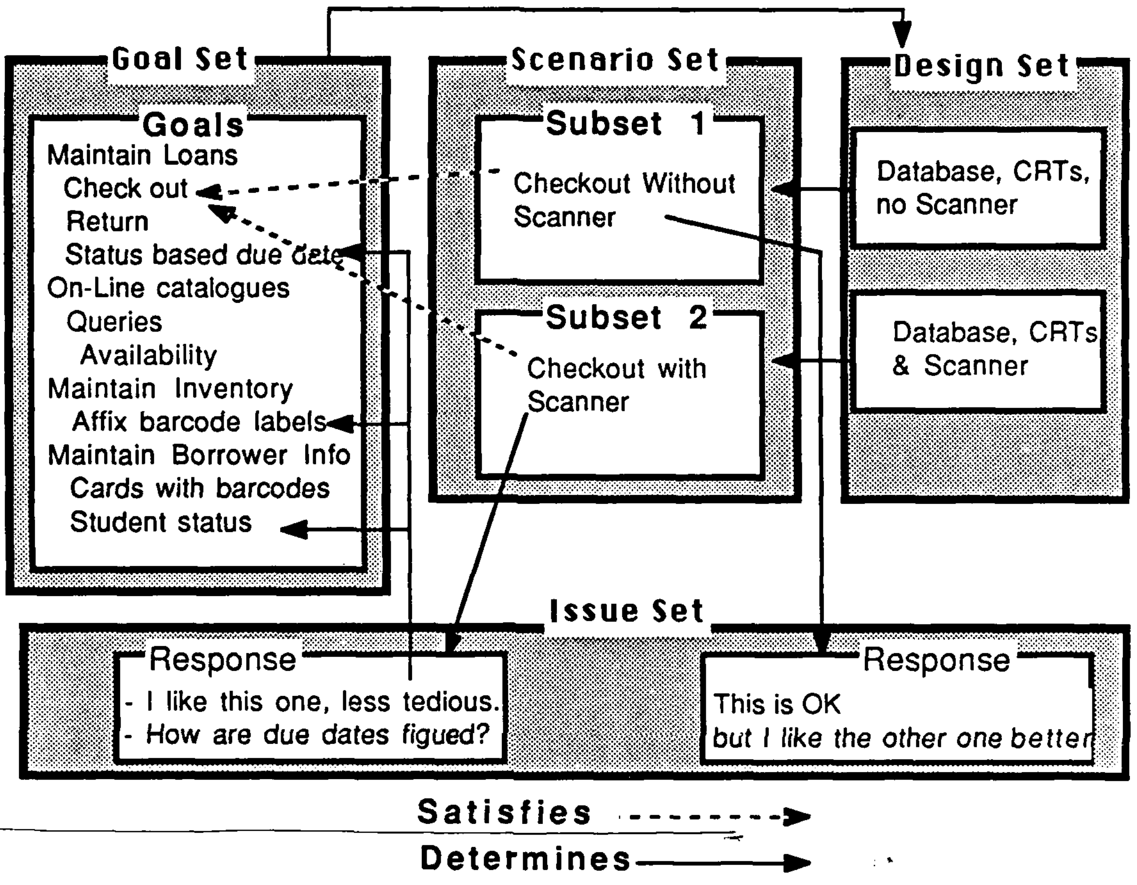


Figure A-2. Architecture after first scenario generation/evaluation cycle

APPENDIX B
DETAIL OF THE GOAL DECOMPOSITION STACK

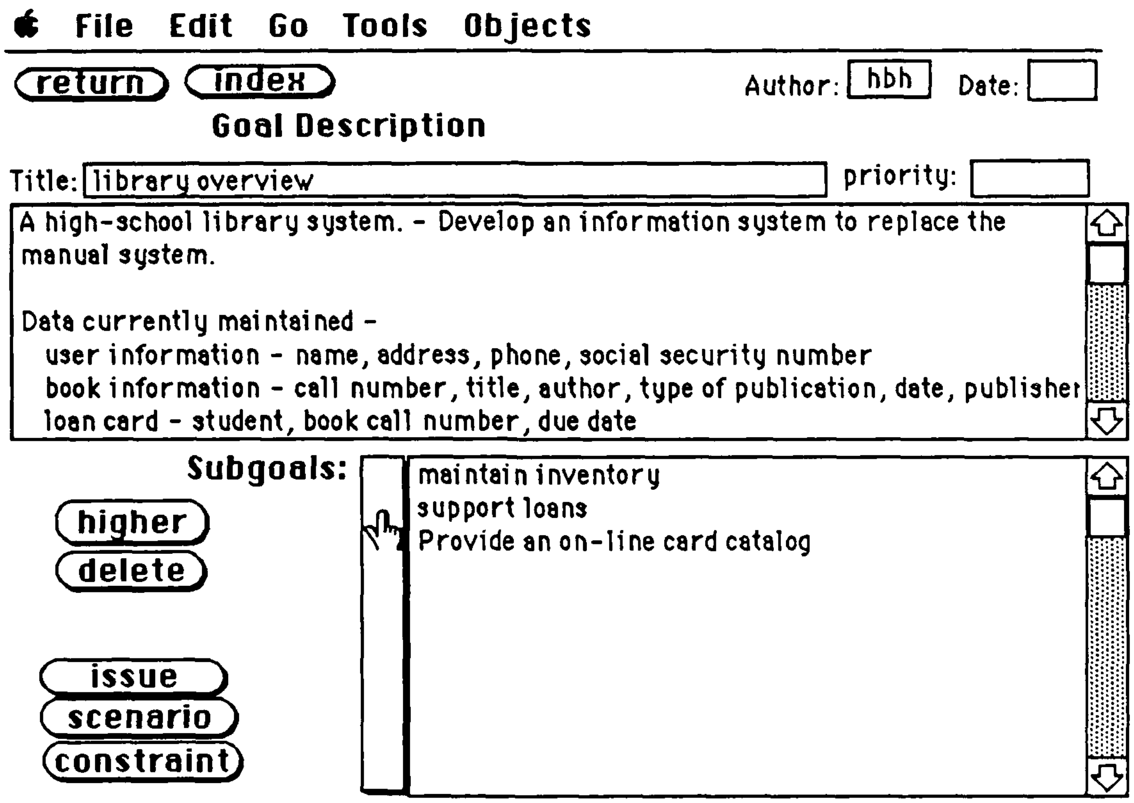
In this Appendix, we explore in some detail the *goal decomposition* stack described in Chapter four as it has been implemented in a prototype SBRE tool. To do so, we will look at all of the components (i.e., cards, fields, buttons, and scripts) that make up the stack. The arrangement of, and functions performed within, this stack are very representative of the those throughout the system. To develop an in-depth understanding of HyperCard, Goodman's book should be reviewed [Go87].

As described in Chapter Four, the goal decomposition stack is made up of *goal description* cards on which the goals for the system are recorded and maintained. The buttons on the card allow for the creation of a hierarchic structure of goals their subgoals. We will first explore this card. We then will look at an overview card which is unique within the stack. Its function is to compile and display an overview of the cards within the stack, formatted to represent the hierarchy. From this card, any card in the stack may be reached using a button.

The Goal Description Card

Illustrated in Figure B-1 is the Goal Description card which makes up the goal hierarchy. The card is used to record and maintain the detailed description of a particular goal. Additionally, it displays the subgoals that directly support the goal described. This card is typical of the ones storing information throughout the SBRE system. The main characteristics for such a card are a description title for indexing purposes, and a detailed description in a scrolling field. We will now take a closer look at the components that make up this card.

There are fourteen *fields* associated with the goal description card, eight of which are hidden and two of which are scrolling. Within these fields are the information that distinguishes the card from any other in the stack. Scrolling fields are used to record varying amounts of information that may not fit completely on the card. Hidden fields contain information associated with the card that remains hidden from view. Additionally, fields can be implemented as arrays by using the lines of the field as an index.



Author: hbh Date:

Title: library overview priority:

```
A high-school library system. - Develop an information system to replace the
manual system.

Data currently maintained -
  user information - name, address, phone, social security number
  book information - call number, title, author, type of publication, date, publisher
  loan card - student, book call number, due date
```

```
maintain inventory
support loans
Provide an on-line card catalog
```

Figure B-1. The Goal Description Card

-- The fields in the goal description card are identified below along with a brief description of their content.

Scrolling Field "goalText"	-- the detailed description of the goal in a scrolling field
Scrolling Field "subgoals"	-- the titles of the supporting subgoals in a scrolling field
Field "goalDesc"	-- the title of the goal
Field "goalAuth"	-- the initials of the person who entered the goal
Field "goalDate"	-- the date the goal was entered
Field "priority"	-- if desired, establish a priority for this goal

-- The following fields are "hidden" fields which offer a way of storing information on a field without showing it on the display:

Hidden Field "children"	-- the card numbers of the subordinate goals in this stack
Hidden Field "parent"	-- the card numbers of the superordinate goal in this stack
Hidden Field "scenario names"	-- the titles of any scenarios associated with the goal
Hidden Field "scenario cards"	-- the location of the scenarios' begin cards
Hidden Field "constraint names"	-- the titles of any constraints
Hidden Field "constraint cards"	-- the card numbers of those constraints in the Constraints stack
Hidden Field "issue names"	-- the titles of any issues associated with the goal description
Hidden Field "issue cards"	-- the card numbers of those issues within the issues set

The following is a list of the buttons associated with the goal description cards. They represent the actions that can be taken while a goal description card is on the screen. The actions for these buttons are controlled in the stack script that follows. The script is not associated directly with the buttons themselves to save space. A script in HyperCard is event-driven. An event such as a MouseUp (a click on the mouse) passes a message up a hierarchy of HyperCard objects where it activates a script if is encountered. The following buttons have no scripts associated with them, so the message passes on to the stack script described later.

Button "delete"	-- delete this goal description card and all subordinate goals
Button "higher"	-- go to the superordinate goal
Button "scenario"	-- create, review, or delete the scenarios associated with this goal
Button "issue"	-- create, review, or delete the issues associated with this goal
Button "subgoals"	-- create, review, or delete the sugoals associated with this goal (this is the button on which the browse tool is sitting in Figure B-1)
Button "index"	-- go to the stack overview index
Button "constraint"	-- create, review, or delete the constraints associated with this goal
Button "return"	-- this button has the following script associated with it

Script:

```
  on mouseUp
    pop card              -- go to the card that has been previously pushed into a stack
  end mouseUp
```

This script is associated with the stack, it acts as a handler for the buttons on the goal description cards throughout the stack. Since there are no scripts associated with these buttons or the cards, the MouseUp event passes a message to the stack level where it is captured by this script. The reason for doing this is that it is more efficient that associating scripts with every button in the stack.

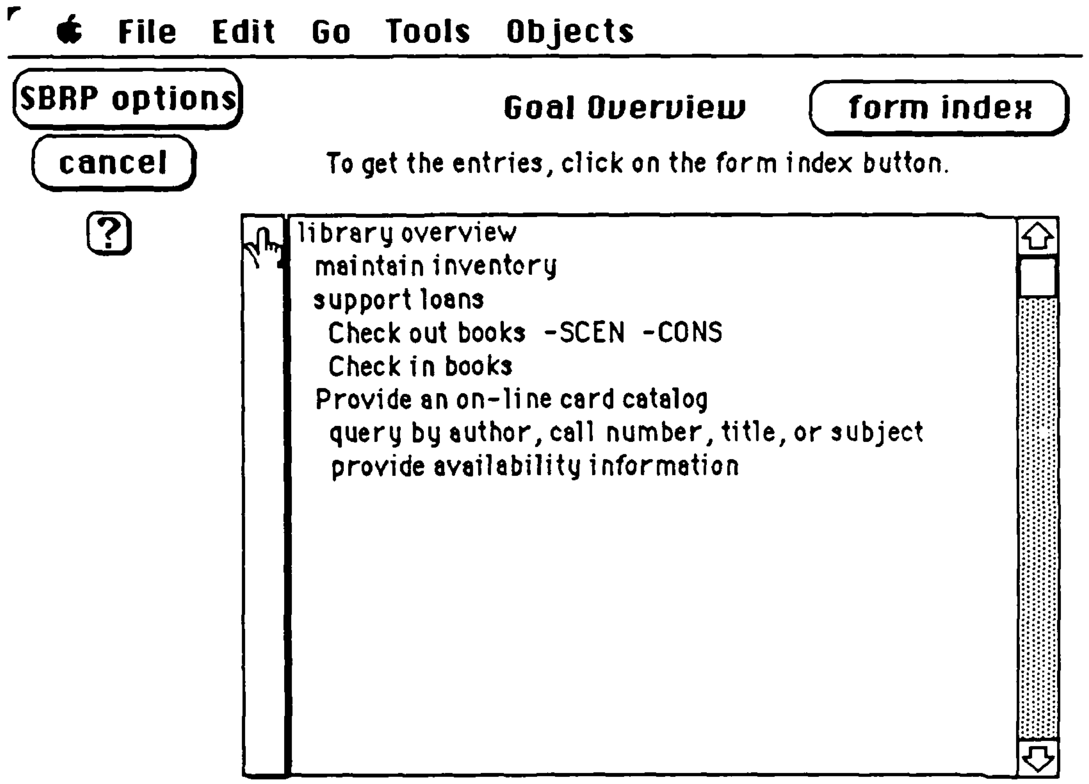
Stack Script:

```
  on MouseUp                                  -- MouseUp is an event that activates a script
    if the target contains "button" then      -- target identifies that the event originated
with a button
      put the short name of the target into ButtName   -- store the name of the button
      ChooseAction Buttname              -- call a subroutine and pass Buttname as a parameter
    end if
  end MouseUp
```

```
on ChooseAction ButtName
 if word 1 of ButtName is "Subgoals" then          -- the "Subgoal" button was clicked
-- calculate the line number of the desired entry
   put item 2 of rect of background button "subgoals" into start
   subtract the scroll of field "subgoals" from start
   put 1 + (item 2 of the ClickLoc - start) div 13 into LineNum
   DoSubGoal LineNum     -- perform "DoSubGoal" passing the parameter "LineNum"
 end if
 if word 1 of ButtName is "higher" then    -- the "higher" button was clicked
   GoHigher
 end if
 if word 1 of ButtName is "scenario" then          -- the "scenario" button was clicked
   DoScenario
 end if
 if word 1 of ButtName is "constraint" then        -- the "constraint" button was clicked
   DoConstraint
 end if
 if word 1 of ButtName is "issue" then     -- the "issue" button was clicked
   DoIssue
 end if
 if word 1 of ButtName is "delete" then    -- the "delete" button was clicked
   DoDelete
 end if
 if word 1 of ButtName is "index" then     -- the "index" button was clicked
   go to card "general goal index"
 end if
```

```
  end ChooseAction

on DoSubGoal LineNum
-- this is a subroutine that processes the subgoals for a goal description
  if line lineNum of field "subgoals" is empty then
        -- fields can be manipulated like arrays by using "line" numbers as an index
   answer "no subgoal specified" with "OK"
   exit DoSubGoal
  end if
  if line LineNum of field "children" is empty then
   answer "Create a detail for this subgoal?" with "Cancel" or "OK"
   if it is "cancel" then
    exit DoSubGoal
   else
    set the lockScreen to true
    CreateSub LineNum
   end if
  end if
  go to card id (line lineNum of field "children")
end DoSubGoal

on CreateSub LineNum
-- a subroutine to create a new subgoal.  This creates and links a new card.
  global UserInits
  put word 3 of id of this card into linkBack          -- store the name of the current card
  put line LineNum of field "SubGoals" into SubGoal
  doMenu "New Card"
  put SubGoal into field "GoalDesc"
  put UserInits into field "GoalAuth"
  put the date into field "GoalDate"
  put LinkBack into field "parent"
  put word 3 of id of this card into LinkDown
  go to card id LinkBack
  put LinkDown into line LineNum of field "children"
end CreateSub

on goHigher   -- this subroutine goes to the superordinate goal (the parent)
  if field "parent" is empty then
    answer "This is the root" with "OK"
  else
    go to card id (field "parent")
  end if
end goHigher

on DoDelete   -- This subroutine deletes a description card and all of its subordinate goals
  if field "parent" is empty then
   answer "Cannot delete the root goal" with "OK"
   exit DoDelete
  end if
  answer "Delete this goal and all subgoals?" with "Ok" or "cancel"
  if it is "OK" then
    set LockScreen to true      -- LockScreen is a system function to freeze the display
    set the cursor to 4         -- displays a watch type cursor to indicate processing
    ProcDelete
  end if
end DoDelete


on ProcDelete              -- This subroutine carries out the actual deletion of the cards
  if field "children" is empty then
    put word 3 of id of this card into LinkBack
    put field "parent" into goParent
    doMenu "Delete Card"
    go to card id goParent
    FixParent LinkBack
  else
    repeat until field "children" is empty
      go to card id (line 1 of field "children")
      ProcDelete
    end repeat
    ProcDelete
  end if
end ProcDelete


on FixParent LinkBack          -- This subroutine maintains the links after a deletion
  put 1 into count1
  repeat until line count1 of field "subgoals" is empty
    if line count1 of field "children" is LinkBack then
      delete line count1 of field "children"
      delete line count1 of field "subgoals"
    else
      add 1 to count1
    end if
  end repeat
end FixParent

on DoScenario
-- this subroutine maintains the scenarios associated with a goal description, it does so by
passing the names and locations of the scenarios linked to this goal to an index card in the
scenario stack where the scenarios exist and the maintenance is performed.
  if field "scenario cards" is empty then
    answer "No scenarios exist, create?" with "yes" or "cancel"
    if it is "cancel" then
      exit DoScenario
    end if
  end if
  global HoldScNames, HoldScCards, LinkBack, GoalTitle
  put field "scenario Names" into HoldScNames
-- move the hidden fields into global variables
  put field "scenario cards" into HoldScCards
  put field "GoalDesc" into GoalTitle
  put word 3 of id of this card into LinkBack
  go to card "goal scenario index" of stack "scenarios"
end DoScenario

on DoConstraint
-- this subroutine maintains the constraints (see comments on DoScenarios)
    if field "constraint cards" is empty then
      answer "No constraints exist, create?" with "yes" or "cancel"
    if it is "cancel" then
     exit DoConstraint
    end if
  end if
  global HoldScNames, HoldScCards, LinkBack, GoalTitle
  put field "constraint Names" into HoldScNames
  put field "constraint cards" into HoldScCards
  put field "GoalDesc" into GoalTitle
  put word 3 of id of this card into LinkBack
  go to card "goal constraint index" of stack "constraints"
end DoConstraint

on DoIssue
-- this subroutine maintains the issues associated with this goal description card
  (see comments on DoScenarios)
  if field "issue cards" is empty then
    answer "No issues exist, create?" with "yes" or "cancel"
    if it is "cancel" then
     exit DoIssue
    end if
  end if
  global HoldScNames, HoldScCards, LinkBack, GoalTitle
  put field "Issue Names" into HoldScNames
  put field "Issue cards" into HoldScCards
  put field "GoalDesc" into GoalTitle
  put word 3 of id of this card into LinkBack
```

```
    go to card "goal issue index" of stack "issues"
end DoIssue
```



Goal Overview

To get the entries, click on the form index button.

```
library overview
 maintain inventory
 support loans
  Check out books  -SCEN  -CONS
  Check in books
 Provide an on-line card catalog
  query by author, call number, title, or subject
  provide availability information
```

Figure B-2. An overview for browsing the goal stack

The Goal Overview Card

We now will describe how a browsing capability is provided in HyperCard using an outline of the goal titles. The card illustrated in Figure B-2 is the goal *overview card* in the prototype system. From it, any card can be accessed from within the goal decomposition stack. To do so, the index must be compiled by clicking on the "form index button", this will display the entries in the scrolling field (this script will be described) In addition, information is provided about links to the card, indicated by -SCEN (scenarios), -

CONS (constraints), or -ISS (issues). To get to a particular goal description card, the user clicks on the button to the left of, and even with the entry in the overview.

-- Associated with this card is a small script to empty the field when the card is closed.

Card Script:

```
   on CloseCard
     put empty into card field "index"
   end CloseCard
```

-- There are three fields associated with this card.

Scrolling Field "index" -- displays the outline generated in a scrolling field

Hidden Field "entries" -- holds the corresponding locations for the items displayed in the outline

Hidden Field "info" -- contains the information displayed by activating the "help" button

-- The buttons associated with this card have their scripts associated with the card since there is only one card. The buttons are described along with their scripts.

Button "index button" -- this is the long button to the left of the overview's scrolling field where the browse tool is situated (Figure B-2)

Script:

```
   on mouseUp
-- calculate the line number of the desired entry in the outlines field which is the same as the
line number of the hidden field entries.
     put item 2 of rect of card button "index button" into start
     subtract the scroll of card field "index" from start
     put 1 + (item 2 of the ClickLoc - start) div 13 into LineNum
     if line LineNum of card field "index" is empty then
```

```
    answer "Cant go to a nameless goal" with "OK"
  else
    push card
    go to card id (line LineNum of card field "Entries")
  end if
end mouseUp
```

Button "cancel"   -- cancel this card, go back to card last pushed

Script:

```
on mouseUp
  pop card
end mouseUp
```

Button "SBRP options"   -- return to the SBRE tool's main menu

Script:

```
on mouseUp
  go to card "options" of "SBRP"
end mouseUp
```

Button "help"   -- the question mark button displays help information by showing the hidden field *info*

Script:

```
on mouseUp
  set visible of card field "info" to not the visible of card field "info"
end mouseUp
```

Button "form index" -- compile the overview of titles, locations, and link information for the goal description cards in the stack

Script:

```
on mouseUp
  global HoldTitles, HoldEntries, LastLine, RecurLevel
  put empty into HoldEntries
  put empty into HoldTitles
  put empty into card field "index"
  put 1 into LastLine
  put 0 into RecurLevel
  set the lockScreen to true
  set the cursor to 4
  go to card "root goal"
  FormIndex
  go to card "general goal index"
  if HoldTitles is empty then
    answer "No entries found" with "OK"
  else
    put HoldTitles into card fieid "index"
    put HoldEntries into card field "Entries"
end if
end mouseUp

on FormIndex        -- recursively process the nodes in the heirarchy
  global HoldTitles, HoldEntries, LastLine, RecurLevel
  put field "GoalDesc" into line LastLine of HoldTitles
  put word 3 of id of this card into line LastLine of HoldEntries
```

```
    repeat for RecurLevel
    put "  " before word 1 of line LastLine of HoldTitles      -- indent the entry to indicate
level
    end repeat
    if not (field "scenario cards" is empty) then        -- if scenario links are found, indicate
so put the number of words of line LastLine of HoldTitles into NumWords
      add 1 to NumWords
      put "  -SCEN" after word NumWords of line LastLine of HoldTitles
    end if
    if not (field "constraint cards" is empty) then       -- indicate if constraint links were
found
      put the number of words of line LastLine of HoldTitles into NumWords
      add 1 to NumWords
      put "  -CONS" after word NumWords of line LastLine of HoldTitles
    end if
    if not (field "issue cards" is empty) then  -- indicate if issue links were found
      put the number of words of line LastLine of HoldTitles into NumWords
      add 1 to NumWords
      put "  -ISS" after word NumWords of line LastLine of HoldTitles
    end if
    put word 3 of id of this card into HoldPlace
    add 1 to LastLine
    if field "children" is empty then   -- exit the routine if no subgoals exist for this goal
      exit FormIndex
    else
      add 1 to RecurLevel              -- otherwise call the routine again to process the
subgoals
```

```
    put 1 into LineCount
    repeat until line LineCount of field "children" is empty
      go to card id (line LineCount of field "children")
      FormIndex
      add 1 to LineCount
      go to card id HoldPlace
    end repeat
    subtract 1 from RecurLevel
  end if
end FormIndex
```

REFERENCES

Ac67 Ackoff, Russell L. (1967), Management Misinformation Systems, Management Science, 14 (4), 147-156.

Al84 Alavi, Maryam (1984), An Assessment of the Prototyping Approach to Information Systems Development, Communications of the ACM, 27 (6), 556-563.

Am76 Amkreutz, J. H. A. E. (1976), Cybernetic Model of the Design Process, Computer Aided Design, 8 (3), 187-191.

An83 Andrews, William C. (1983), Prototyping Information Systems, Journal of Systems Management, 34 (9), 16-18.

Ba77 Balzer, R., Goldman, N., and Wile, D. (1977), Informality in Program Specifications, IEEE Transactions on Software Engineering, SE-4 (2), 94-103.

Bo84 Boehm, Barry W., Gray, Terrence E., and Seewaldt, Thomas (1984), Prototyping Versus Specifying: A Multiproject Experiment, IEEE Transactions on Software Engineering, SE-10 (3), 290-302.

Bo78 Boland, Richard J. Jr. (1978), The Process and Product of System Design, Management Science, 24 (9), 887-898.

Bo71 Bordon, George A. (1971), An Introduction to Human-Communication Theory, Dubuque, Iowa: Wm. C. Brown Company Publishers.

Br75 Brooks, F. P. Jr. (1975), The Mythical Man-Month, Reading, Mass.: Addison-Wesley.

Br87 Brooks, F. P. Jr. (1987), No Silver Bullet, Essence and Accidents of Software Engineering IEEE Computer, April 1987, 10-19.

Ca79 Carroll, John M., Thomas, John C., and Malhotra, Ashok (1979), Clinical-Experimental Analysis of Design Problem Solving, Design Studies, 1 (2), 84-92.

Ca80 Carroll, John M., Thomas, John C., Miller, Lance A., and Friedman, Herman P. (1980), Aspects of Solution Structure in Design Problem Solving, American Journal of Psychology, 93 (2), 269-284.

Co86 Conklin, Jeff (1986), A Theory and Tool for Coordination of Design Conversations, MCC Technical Report STP-236-86, Microelectronics and Computer Technology Corporation, Austin, Tex.

Co87 Conklin, Jeff (1987), Hypertext: An Introduction and Survey, IEEE Computer, 20 (9), 17-41.

Co88 Conklin, Jeff, and Begeman, Michael (1988), gIBIS: A Hypertext Tool for Team Design Deliberation (extended abstract), MCC Technical Report STP-016-88, Microelectronics and Computer Technology Corporation, Austin, Tex.

Co85 Conklin, Jeff, and Richter, Charles (1985), Support for Exploratory Design, MCC Technical Report STP-117-85, Microelectronics and Computer Technology Corporation, Austin, Tex.

Cr82 Crowley, D. J. (1982), Understanding Communication: The Signifying Web, New York: Gordon and Breach Science Publishers.

Da82 Davis, G. B. (1982), Strategies for Information Requirements Determination, IBM Systems Journal, 21 (1), 4-30.

Fl83 Floyd, Christiane (1983), A Systematic Look at Prototyping, Approaches to Prototyping, Proceedings of the Working Conference on Prototyping, Namur, Belgium.

Gl82 Gladden, G. R. (1982), Stop the Life-Cycle, I Want to Get Off, ACM SIGSOFT Software Engineering Notes, 7 (2). 35-39.

Go83 Gomaa, Hassan (1983), The Impact of Rapid Prototyping on Specifying Users Requirements, ACM SIGSOFT Software Engineering Notes, 8 (9), 17-27.

Go87 Goodman, Danny (1987), The Complete HyperCard Handbook, New York: Bantam Books, Inc.

Gu87 Guindon, R., Curtis, Bill, and Krasner, Herb (1987), A Model of Proccesses in Software Design: An Analysis of Breakdowns in Early Design Activities by Individuals, MCC Technical Report #STP-283-87, Microelectronics and Computer Technology Corporation, Austin, Tex.

Ho82 Hooper, James W., and Hsia, Pei (1982), Scenario-Based Prototyping for Requirements Identification, ACM SIGSOFT Software Engineering Notes, 7 (5), 88-92.

IE84 IEEE Guide to Software Requirements Specifications, ANSI/IEEE Std. 830-1984, New York: Institute of Electrical and Electronics Engineers, Inc.

Je84 Jenkins, A. M., Naumann, Justus D., and Wetherbe, James C. (1984), Empirical Investigation of Systems Development Practices and Results, Information and Management, 7, 73-82.

Jo83 Johnson, James R. (1983), A Prototypical Success Story, Datamation, 29 (11), 251-256.

Ka88 Kaufman, L. D. (1988), Scenario Selection and Implementation Techniques for Scenario-Based Rapid Prototyping, SERC-TR-19-F, Computer and Information Sciences Department, University of Florida, Gainesville, FL.

Kl86 Klapp, Orrin E. (1986), Overload and Boredom: Essays on the Quality of Life in the Information Society, New York: Greenwood Press.

La84 Langle, Gernot B., Leitheiser, Robert L., and Naumann, Justus D. (1984), A Survey of Applications Systems Prototyping in Industry, Information & Management, 7, 273-284.

Le82 Lehman, M. M. (1982), The Role of Executable Metric Models in the Programming Process, Proc. ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping, Columbia, MD.

Le87 Leite, Julio Cesar S. P. (1987), A Survey on Requirements Analysis, Advanced Software Engineering Project, RTP-071, University of California at Irvine.

Ma80 Malhotra, Ashok, Carroll, John M., Thomas, John C., and Miller, Lance A. (1980), Cognitive Processes in Design, International Journal Man-Machine Sudies, 12, 119-140.

Ma82 Mason, R. E. A., Carey, T. T., and Benjamin, A. (1982), Act/1: A Tool For Information Systems Prototyping. ACM SIGSOFT Software Engineering Notes, 7 (5), 120-126.

Mc87 McVay, Monte (1987), Models of Systems Development and Design, Unpublished.

My85 Myers, Ware (1985), MCC: Planning the Revolution in Software, IEEE Softwarc, November 1986, 68-73.

Ne72 Newell, Alan and Simon, Herbert A. (1972), Human Problem Solving, Englewood Cliffs, N.J.: Prentice-Hall.

Ri73 Rittel, Horst W. J. and Webber, Melvin M. (1973), Dilemmas in a General Theory of Planning, Policy Sciences, 4, 155-169.

Sa87 Sasso, William C., and McVay, Monte (1987), The Constraints and Assumptions Interpretation of Systems Design: A Descriptive Process Model, Center fo Research on Informations Systems, Graduate School of Business Administration, New York University.

Sc81 Scharer, Laura (1981), Pinpointing Requirements, Datamation, 27 (4), 139-151.

Sc74 Scott, R. F., and Simmons, D. B. (1974), Programmer Productivity and the Delphi Technique, Datamation, 20 (5), 71-73.

Si73 Simon, Herbert A. (1973), The Structure of Ill Structured Problems, Artificial Intelligence, 4, 181-201.

So85 Sommerville, Ian (1985), Software Engineering, Reading, Mass.: Addison-Wesley Publishing Co.

St83 Steele, Anne C., and Nowell, Barbara J. (1983), Conceptual Prototyping, 1983 ACM Annual Conference, 226-228.

Ta82 Taylor, Tamara, and Standish, Thomas A. (1982), Initial Thoughts on Rapid Prototyping Techniques, ACM SIGSOFT Software Engineering Notes, 7 (5), 160-166.

Tu87 Turner, Gary Stephen (1987), Prototyping: A Better Way To Develop Software, Master's Thesis, Georgia Institute of Technology.

Vi83 Vitalari, Nicholas P., and Dickson, Gary W. (1983), Problem Solving for Effective Systems Analysis: An Experimental Exploration, Communications of the ACM, 26 (11), 948-955.

We49 Weaver, Warren (1949), The Mathematics of Communication, Scientific American, 181 (1), 11-15.

We87 Wexelblat, Alan (1987), Report on Scenario Technology, MCC Technical Report STP-139-87, Microelectronics and Computer Technology Corporation, Austin, Tex.

Yo87 Young, Richard M., and Barnard, Phil (1987), The Use of Scenarios in Human-Computer Interaction Research: Turbocharging the Tortoise of Cumulative Science, CHI + GI 87 Human Factors in Computing Systems and Graphics Interface, Toronto, 291-296.

BIOGRAPHICAL SKETCH

Hilliard Baxter Holbrook III [redacted], the son of Commander Hilliard Baxter Holbrook II, the grandson of Rear Admiral Hilliard Baxter Holbrook, and the grandson of Godfrey [redacted], a Swiss [redacted] Typical of many military offspring, he attended six different schools before graduating from high school in 1971. He was then employed as a carpenter in Virginia Beach before enlisting in the United States Air Force in 1975. Upon completion of basic training, he was trained as an explosive ordnance disposal specialist and served at Eglin AFB, Florida, for five years. It was there he met, and was coerced into marriage by, the former Jane Doherty. In 1980, the Air Force sent then Staff Sergeant Holbrook to the University of Florida to pursue a bachelor's degree in computer science. From there, he attended Officer Training School and was commissioned a second lieutenant on August 4, 1982. He then spent four years as a computer systems analyst at the Air Force Data Systems Design Office at Gunter AFS, Alabama. During this assignment, he was selected to pursue his master's degree in computer science at the University of Florida. Upon graduation, Captain Holbrook will serve as an instructor for computer science at the US Air Force Academy.