394

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE SOFTWARE ENGINEERING PROTOTYPE

by

Michael R. Kirchner

June 1983

Thesis Advisor: Gordon C. Howell

Approved for public release; distribution unlimited

T210117

Monterey, CA 93943

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Software Engineering Prototype		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June, 1983
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael R. Kirchner		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California		12. REPORT DATE June, 1983
		13. NUMBER OF PAGES 100
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

software engineering, software prototype, software design, design theories, software engineering environments, case studies, software development, information systems development, system development life cycle

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

Experience has shown that the traditional method of software development often has poor results. Recently, a new approach to software development, the prototype approach, has been proposed. This thesis presents an integrated view of general design theories and relates that view to software design and development. The current thought on prototypes is described and the basic requirements for a software engineering environment are presented. (Cont)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

 SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued) Block # 20

Software prototypes are shown to support the integrated view of designs. Four case studies of using prototypes are presented and recommendations for further study are made.

S/N 0102- LF- 014- 6601

Approved for public release; distribution unlimited.

The Software Engineering Prototype

by

Michael R. Kirchner
B.S., Illinois Benedictine College, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
June 1983

ABSTRACT

Experience has shown that the traditional method of software development often has poor results. Recently, a new approach to software development, the prototype approach, has been proposed. This thesis presents an integrated view of general design theories and relates that view to software design and development. The current thought on prototypes is described and the basic requirements for a software engineering environment are presented. Software prototypes are shown to support the integrated view of design. Four case studies of using prototypes are presented and recommendations for further study are made.

TABLE OF CONTENTS

I. INTRODUCTION 10

II. MODELS OF DESIGN METHODS 12

A. STRUCTURED MODELS OF DESIGN 12

B. WICKED PROBLEMS 15

C. ACCUMULATED KNOWLEDGE MODELS OF DESIGN 18

1. Design is Argumentative 18

2. Patterns in Design 18

3. Design as Learning 19

4. Design is Satisficing 20

D. DESIGN AS A TECHNOLOGICAL ACTIVITY 21

E. DESIGN IS EVOLUTIONARY 21

F. SUMMARY 23

III. SOFTWARE DESIGN METHODS 25

A. SOFTWARE DESIGN IS SYMMETRICAL AND ADAPTIVE . 25

B. DESIGN IS SATISFICING 26

C. SOFTWARE DESIGN IS A WICKED PROBLEM 28

D. COMMUNICATIONS BETWEEN THE DESIGNER AND THE END USER 33

E. SOFTWARE DESIGN IS LEARNING 35

F. SOFTWARE DESIGN HAS AN ORGANIZATIONAL CONTEXT 40

G. SOFTWARE DESIGN IS EVOLUTIONARY 43

H. SUMMARY 47

IV. THE SOFTWARE PROTOTYPE 49

A. INTRODUCTION 49

B. THE PROTOTYPE PROCESS 50

C. PROTOTYPES AS MODELS 51

D. STRATEGIES TO PRODUCE PROTOTYPES 53

1. The 'Methodology' Strategy 53
2. Executable Specifications 53
3. Automatic Programming 54
E. USES OF PROTOTYPES 55
1. To Clarify the User's Requirements 55
2. To Verify the Feasibility of Design . . . 56
3. To Create the Final System 56
F. PROTOTYPES ADDRESS THE ESSENTIAL DESIGN ELEMENTS 57
1. Prototyping is a Symmetrical and Adaptable Process 57
2. Prototyping 'Tames' the Wicked Problem . . 57
3. Software Prototyping is Satisficing . . . 59
4. Prototyping is Communicating 59
5. The Software Prototype is a Learning Aid . 60
6. The Prototype Processs Accounts for Organizational Issues 61
7. The Prototype Process is Evolutionary . . 62
G. SUMMARY AND INTERMEDIATE CONCLUSIONS 63

V. THE SOFTWARE ENGINEERING ENVIRONMENT 65
A. INTRODUCTION 65
B. CHARACTERISTICS OF SOFTWARE ENGINEERING ENVIRONMENTS 66
1. Development Support Tasks 66
2. Integrated 67
3. Uniform 67
4. Support a Solution Strategy 67
5. Adaptable 68
6. Functionally Unique 68
7. Interactive 68
8. Recent Developments 68
C. A SOFTWARE ENGINEERING ENVIRONMENT FOR PROTOTYPES 69

1. Technical Components 69
2. Support for Software Design 72
3. Support for the Prototype Process 74
D. SUMMARY 76

VI. CASE EXAMPLES 77
A. SYMMETRY, EVOLUTION, SATISFICING, AND COMMUNICATION 77
B. LEARNING 79
C. WICKED PROBLEMS, COMMUNICATIONS, AND THE ORGANIZATIONAL CONTEXT 80
D. COMMUNICATION, LEARNING, AND EVOLUTION 81
E. SUMMARY 82

VII. CONCLUSIONS 84

VIII. RECOMMENDATIONS FOR FURTHER STUDY 86
A. MANAGEMENT 86
B. ACQUISITION AND CONTRACT MANAGEMENT 86
C. ORGANIZATIONAL CONTEXT 87
D. QUALITY 87
E. REPRESENTATION 88

LIST OF REFERENCES 89

INITIAL DISTRIBUTION LIST 99

LIST OF TABLES

I. Design Methodclogies 31
II. Hypotheses Tested in the Experiment 34
III. Results of the Experiment 34

LIST OF FIGURES

2.1 Alexander's Design Phases 14

3.1 Kolb's Learning Cycle Model 37

3.2 A Constructive Conflict Model for User Involvement 39

3.3 Typical Life Cycle Representation 45

4.1 The Prototype Model 52

4.2 Evolution of Prototypes 63

I. INTRODUCTION

Current software engineering practices are based on a development model which is 10 to 15 years old., This model is often referred to as the waterfall model. The waterfall model shows the development of software as a series of discrete steps [Ref. 1, 2, 3, 4, and 5].

Experience indicates, however, that software development is not as discrete as the model indicates, so the model has been refined by adding loops between each of the steps. Furthermore, as software maintenance has gained recognition, there is increased pressure to refine the waterfall model to show the added importance of maintenance in the software life-cycle.

The software engineering profession's concern about software maintenance, which is more properly termed refinement and enhancement, has prompted several conjectures. Dodd [Ref. 20] has suggested that the current cycle of develop, implement, refine and enhance, implement, refine and enhance, implement, and so on is really the construction and refinement of a prototype system.

Several other authors have suggested that we should develop software prototypes as an alternative to the traditional, or waterfall, approach to software development [Ref. 68, 36, 62]. Their principal argument is that the process of software development is really iterative, slowly expanding toward a completed system. Other reasons include enhanced communications between the user and designer, fewer requirements problems, quicker turnaround between initial system need and initial system implementation, to name a few.

The process of developing a software prototype has significant intuitive appeal for users and managers; they can try a system out before committing themselves to a system which is either unsatisfactory or undelivered. Aside from this appeal and the benefits often cited, there seems to be little discussion about the principles underlying the development of software prototypes.

This thesis presents one view of how the process of developing software prototypes supports some basic elements of general design theory and software design specifically. Chapter II develops an integrated set of design elements based on several published models of the general design process. Chapter III relates these design elements to software development by citing examples from the computing and information science literature. The purpose is to show that software design is similar to other fields of design.[1]

Chapter IV introduces the software prototype. The process of developing software prototypes, their roles as models, construction strategies, and the principal uses of prototypes are described. The chapter concludes by showing how prototypes support the design elements from Chapters II and III. Chapter V briefly describes the essential features of software engineering environments, especially those features which are needed for developing software prototypes. Chapter VI presents four case examples which illustrate the process of developing a software prototype. These cases were chosen because in each of them there was an explicit decision to use prototypes. Chapters VII and VIII present Conclusions and Recommendations for Further Study.

----

[1]To paraphrase Gertrude Stein: Design is design is design is design.

II. MODELS OF DESIGN METHODS

A. STRUCTURED MODELS OF DESIGN

The ideas about design and design methods have undergone some significant changes in the last 20 years. The early models placed their emphasis on the process of design. These models had a rational, discrete notion of design in which the design process was thought to be a sequence of well-defined, highly structured activities. Many theorists applied the ideas and principles of the scientific method to the process. Alexander [Ref. 6] was one of the earliest of the design theorists to carefully explain design. His three most significant contributions were:

1. The symmetry of the design problem--that is, design has two symmetical parts, the <u>form</u> (the solution to the problem) and the <u>context</u> (the setting which defines the problem). "... adaptation is a mutual phenomenon referring to the context's adaptation to the form as much as the form's adaptation to it's context ..." The design problem is an effort to achieve "fitness" between the form and it's context. [Ref. 6]
2. The formal decomposition of a set of requirements into successively smaller subunits.
3. The importance of diagrams in design. A diagram, for Alexander, is "[a]ny pattern which, by being abstracted from a real situation, conveys the physical influence of certain demands or forces ..." [Ref. 6: p. 85]

Alexander chose to emphasize the process of decompostion in his early work. This process was divided into two phases, analysis and synthesis.

In analysis, the designer, faced with a problem, derives a mental picture--often vague and unsatisfactory--of the demands of the context, and then decomposes that picture into sets (a mathematical picture). Synthesis begins by developing diagrams (based on the sets), using the diagrams to form a design, and then deriving the form (see Figure 2.1). Alexander also discussed evaluation (he calls it "goodness of fit"). Goodness of fit is determined by one of two criteria, experimental or non-experimental. The experimental criterion is trial and error where "[t]he experiment of putting a prototype form in the context itself is the real criterion of fit." [Ref. 6: p. 21]. The non-experimental criterion is "[a] complete unitary description of the demands made by the context ..." [Ref. 6: p. 21]. Alexander believes that: 1) trial and error is too expensive and too slow and 2) there is no theory which can express "... a unitary description of the varied phenomena of a particular context." [Ref. 6: p. 20]. For these reasons he concentrates on the <u>process</u> of decomposition.[2]

Alexander's structured view was shared by many theorists during the early 1960's. [Ref. 8, 7]. Archer [Ref. 7] thought of design as a goal-directed activity. The goals or objectives of the problem define the properties required in the solution. The details of the design are the designer's decisions about how to implement those properties [Ref. 7: p. 286].

---

[2] Alexander devotes an entire Appendix to the "Mathematical Treatment of Decomposition."

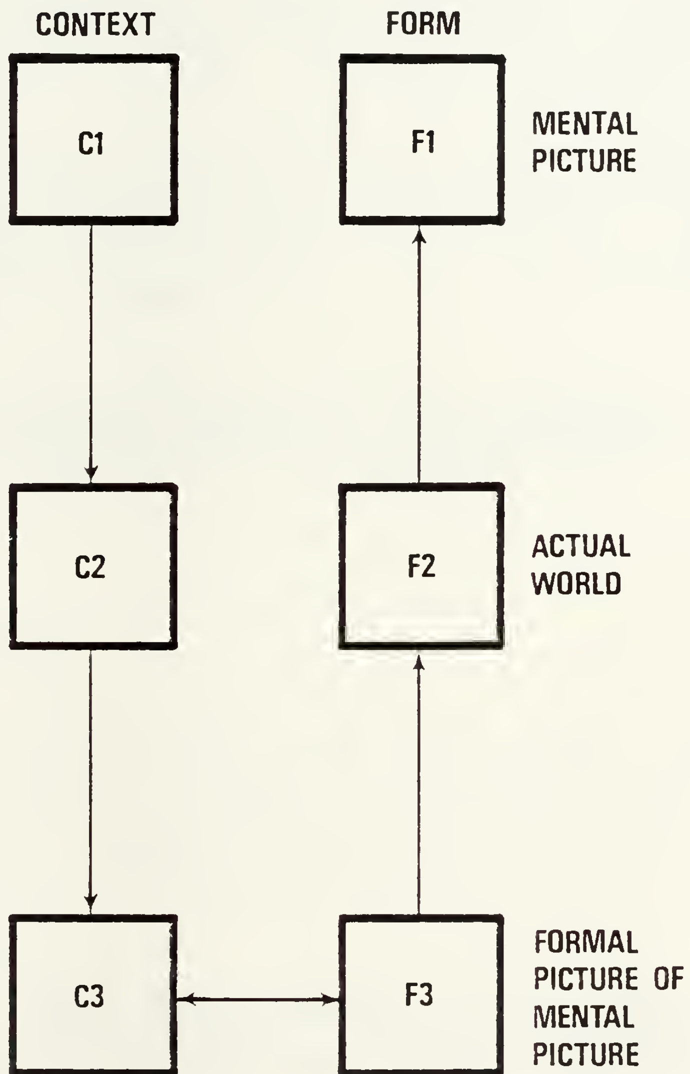

Figure 2.1 Alexander's Design Phases.

Archer identifies three components of the design process:

1. The advance through the project and through time;
2. The branching of the problem into its logical parts; and,
3. A problem-solving process cyclically moving through subproblems (using a 30-step reiterative operational model).

Jones [Ref. 8] called the three stages in his view of the design process divergence, transformation, and convergence. He was quite convinced that designers should think of these stages as separate:

> ...there is little doubt that their separation is prerequisite to whatever changes of methodology are necessary at each stage before they can be reintegrated to form a process that works well at the systems level. [Ref. 8: p. 64]

B. WICKED PROBLEMS

These early models were often criticized. One critique suggested that design problems are "wicked problems" and are not, therefore, amenable to structured analysis (and decomposition). The term "wicked problem" refers to a

> . . . . class of social system problems which are ill-formulated, where the information is confusing, where there are many clients and decision-makers with conflicting values, and where the ramifications in the whole system are thoroughly confusing. [Ref. 9]

Wicked problems have the following properties :

1. Wicked problems ae ill-formulated. They have no definitive formulation and any formulation will correspond to the formulation of the solution. This means that any time a formulation is made, additional

questions can be asked and more information can be requested. This also means that the information needed to understand the problem is determined by one's idea or plan of a solution. In other words, whenever a wicked problem is formulated there must already be a solution in mind.

2. Wicked problems have no stopping rule. Any time a solution is formulated, it could be improved or worked on more. One can stop only because one has run out of resources, patience, etc. (An architect could keep modifying and improving a design solution forever--he stops because he has exhausted his fee, because the building has to be finally built, or because he has exhausted some other resource.)
3. Solutions to wicked problems cannot be correct or false. They can only be good or bad. (There is no correct or false building: there can only be a "good" building or a "bad" building.)
4. In solving wicked problems there is no exhaustive list of admissable operations. Any conceivable plan, strategy or act is permissable in finding a solution and none can be perscribed as mandatory.
5. For every wicked problem there is always more than one possible explanation. The selection of an explanation depends on the employed world-view; the explanation also determines the solution to the problem. (The high cost of construction of a building may be attributed to the "expensive" design, to the high cost of materials, to the wages demanded by unions, to high interest rates and inflation, etc.)
6. Every wicked problem is a symptom of another "higher level" problem. (If the maintenance of the residence is "too expensive" to its inhabitants, this indicates that there is a problem with the income of its inhabitants.)

7. No wicked problem and no solution to it has a definitive test. In other words, any time any test is "successfully" passed it is still possible that the solution will fail in some other respect. (If large windows are designed for a residence to provide the desired views, the heating of the residence may become too expensive.)
8. Each wicked problem is a "one shot" operation. There is no room for trial and error, and there is no possibility for experimentation. (A house is designed and built--there is no going back to the beginning to redesign and rebuild it.)
9. Every wicked problem is unique. No two problems are exactly alike and no solutions or strategies leading to solutions can readily be copied for the next problem. (Even if two residences are designed for the same family, under the same geographical conditions they will never be identical.)
10. The wicked problem solver has no right to be wrong--he is fully responsible for his action.

If design problems are considered as wicked problems, they are certainly incompatible with the early models of design. The early models clearly separated the problem from its solution. With wicked problems, one cannot "define the problem"--they have no definitive formulation. If one followed the procedures of the early models of design, one should be able to establish when a solution was clearly found. Wicked problems, however, have no stopping rule. Some of the proponents of the early models of design devised tests for design solutions. Alexander argued that trial and error should eventually lead to "good fit"; unfortunately, each time a solution is tried, the problem is also changed.

C. ACCUMULATED KNOWLEDGE MODELS OF DESIGN

1. Design is Argumentative

Other design models were proposed following the criticisms of the early, structured models of design. Rittel [Ref. 13] views the whole design process as sequential problem solving in which the cycles form networks. An essential part of this model is the continuous feedback between the designer and the problems environment. Rittel calls this 'argumentation':

> . . . . the designer [is] arguing toward a solution with himself and with other parties involved in the project. He builds a case leading to a better understanding of what is to be accomplished. In its course, solution principles are developed, evaluated in view of their expected performance and decided upon. The parties commit themselves to specific courses of action and to the risks involved in them. In this way, better formulations of the problem are being developed simultaneously with a clearer and clearer image of the solution. [Ref. 13 : p. 19-20]

If arguments are improved procedurally, their content may improve and the products of the design--design decisions--may also be expected to improve. While 'arguing', the parties may gain new insights about the issue, expand their world-view, modify challenged positions, and learn more about other world-views.

2. Patterns in Design

Alexander introduced the concept of pictoral diagrams in design in 1964 [Ref. 6]. Significantly, Alexander believed that the design diagrams were produced _by formal, rigorous analysis_, a design process founded on mathematical decomposition. Since then, Alexander and others [Ref. 10] have concentrated on the diagrams (or Patterns) rather than the process.

Alexander's patterns are not a result of rigorous analysis. Rather, design is a process of acquiring knowledge and then making decisions which reflect that knowledge. The crucial issue for Alexander is the availability of knowledge. That is, the design decision depends on the accumulated knowledge of the designer. Patterns help to provide the designer with the necessary knowledge to solve the problem. The pattern forms the basis of communication between the designer and the client. A pattern--a diagram of what the designer knows and believes important for the problem--is designed and then passed to the client. The client either accepts or does not accept the pattern. In either case, both the client and the designer gain new knowledge: if the pattern is not accepted, the designer proceeds to change the design.

3. Design as Learning

Bazjanac [Ref. 15] views the design process as formulating the problem and proceeding with a search for the definition of the solution. He emphasizes that the formulation of the problem is not final. The formulation reflects the understanding of the problem, based on the designer's knowledge, _at that time_.

> Any solution ... is already basically determined by the definition of the problem. So the "search for solution" is then the search for the definition of the specific solution which best fits the knowledge the designer has at that time. Once the specific solution is defined it is documented. Documentation may start during the definition of the problem and continue sporadically during the definition of the solution--in fact, all three phases may at times take place simultaneously. The ultimate purpose of the documentation is to communicate the definitions of the problem and the solution; its immediate purpose is to aid the designer in the definition of the problem and the solution--to help him detect new aspects of the problem and the solution and to detect inconsistencies in his view. [Ref. 15]

During the search and redefinition, the designer keeps learning more about the problem and the solution. The designer gains new insights which ultimately lead to a new view-- redefinition. The process (formulate the problem, search for the definition of the solution, document the specific solution) is repetitive. The designer continues to re-define and document new formulations until 1) the incremental gain in knowledge becomes insignificant and cannot change the formulation enough to warrent redefinition, 2) the incremental gain becomes too costly, or 3) the designer exhausts available resources (especially time).

4. Design is Satisficing

As the designer and user learn more about the problem and as the solution becomes clearer, more and more design decisions are negotiated [Ref. 13, 15]. Since these design decisions are reached through compromise, they cannot be called optimal, in the sense of management science and operations research.

Simon [Ref. 14] has introduced the idea of satisficing to describe these kinds of negotiated decisions.

> Normative economics has shown that exact solutions to the large optimization problems of the real world are simply not within reach or sight. In the face of this complexity the real-world business firm turns to procedures that find good enough answers to questions whose best answers are unknowable. ... man is ... a satisficer, a person who accepts "good enough" alternatives, not because he prefers less to more but because he has no choice. [Ref. 14: p. 36]

D. DESIGN AS A TECHNOLOGICAL ACTIVITY

Cross and others [Ref. 16] have proposed a view of design which requires the explicit acknowledgement of the organization's role in design.

> 'Technology' ... clearly denotes more than just hardware, and involves, at the very least, consideration of the organizational systems within which machinery is designed, commissioned, operated and paid for. 'Technological' achievements, whether those of building a major bridge or putting a man on the moon, are as much organizational feats as technical ones. [Ref. 16: p. 198]

These considerations lead to their view that a "satisfactory" definition of technology has the following characteristics:

1. Technology is oriented toward practical tasks.
2. Technology relies on different kinds of organized knowledge, of which scientific knowledge is only one. Craft knowledge, design knowledge, and organizational and managerial skill are others.
3. Technological activity takes place in an organizational context. [Ref. 16: p. 198]

Cross and others devote a great deal of space to highlight the difference between knowing "what to do" (scientific knowledge) and knowing "how to do" (design and craft knowledge). Their main point cannot be ignored: the organization plays as large a role in design as does the individual.

E. DESIGN IS EVOLUTIONARY

The early models of design were frequently criticized for their linear, step-by-step view of design. Page [Ref. 11] warned that the design process is not executed straight from analysis to evaluation:

> ....in the majority of practical design situations, by the time you have produced this and found out that and made a synthesis, you realize you have forgotten to analyze something else here, and you have to go around the cycle and produce a modified synthesis, and so on. In practice, you go around several times.

Ellinger stated that the iterative approach to design "... is paricularly suited to novel projects of some complexity." [Ref. 12: p. vi]

Smithies [Ref. 17] has suggested that there are a number of essential stages in design. The first stage, design analysis, is the statement of the problem, P. The next stage consists of finding one or more tentative solutions, TS. This solution is then criticized, C. When the designer criticizes the solution, he or she admits that the problem statement was inadequate. So, the designer re-states the problem and begins anew.

P1-TS1-C1-P2-...-Pn.

Smithies attributes his views about design to Popper [Ref. 18]. Popper believes that the process or activity of understanding can be represented by a general scheme of <u>problem solving by cojecture and criticism</u>. Popper's scheme, adapted by Smithies, is this:

P1-TT-EE-P2.

P1 is the initial problem statement; TT, the 'tentative theory', is the conjecture. EE, 'error elimination', is the critical examination of the conjecture. P2 is the new problem statement which emerges from the examination. It leads to another attempt, and so on [Ref. 18 : p. 164]. Smithies' design stages and Popper's problem-solving scheme are very much like Polya's [Ref. 19] method for solving problems. Software designers should take note: Polya is a mathematician, Popper is a philosopher, and Smithies is an

architect, yet each approaches the solution to a problem in the same way.

The progress of the designer through these stages is marked by increased knowledge and shifting priorities. Clearly that progress is not linear and should be called evolutionary.

F. SUMMARY

Several points about design have been made in the proceeding sections:

1. Design is symmetrical and adaptive;
2. The interesting (i.e., large, complex) design problems can be considered as wicked problems;
3. Communications with the end user are crucial and depend to a large degree on patterns which bridge the communications barrier between designer and end user;
4. Design is a learning process--each party brings a different perspective to the problem (and the solution!) and leaves (or should leave) with an augmented perspective;
5. Design is satisficing;
6. Design takes place in an organizational context;
7. Design is evolutionary.

The separation of these points should not be misconstrued. Each of these aspects is interrelated and to a certain extent mutually dependent on one another. When we say that design is evolutionary, we also imply that design is symmetrical and adaptive. When we say that design is an organizational activity, we also imply that there will be extensive communication during design. Whenever we try to understand the problem, to learn more about our tentative solution, we are raising a problem of understanding, or posing a higher level problem, which implies that design problems are wicked problems.

This interrelated set of design elements forms the back-drop for the remainder of this work. The following chapter presents evidence from the literature that each of the design elements described above is a factor in software design.

III. SOFTWARE DESIGN METHODS

A. SOFTWARE DESIGN IS SYMMETRICAL AND ADAPTIVE

Several instances in the literature point to the symmetry of the software design problem. That is, the solution not only depends on the problem, but the problem depends on the solution. Solution and problem are not separate issues, rather they are intertwined, much like the figure and ground in a painting or picture. Each depends on the other. Unfortunately, most people associated with software design do not appreciate this point. Peters points out that software designers complain bitterly that requirements are poorly defined while customers and analysts often complain that the design is not responsive to the problem or problems as they see them. [Ref. 23 : p. 67]. Peters wasn't the first to recognize this, though. Podolsky wrote a humorous article in 1977 [Ref. 24] where he states "Peer's Law":

Peer's Law

The solution to a problem changes the problem.

Several other authors [Ref. 25, 26, and 27] have also recognized that the problem definition tends to evolve as the designers try to bound the problem, or modify the requirements. McCracken and Jackson [Ref. 27] have gone so far to say that this dependence is analogous to the Heisenberg Principle: Any system development activity inevitably changes the environment out of which the need for the system arose.

Much effort is currently devoted to requirements definition and yet incompleteness, ambiguity, and poor definitions in requirements documents are often pointed to as the foremost problems facing software designers today. The effort which is spent on completely specifying the user's requirements will gain nothing if software design is adaptive.

McCracken and Jackson believe that systems requirements can never be stated fully in advance. To assert otherwise is to ignore the fact that the development process itself changes the user's perceptions of what is possible, increases insights into the applications environment, and often changes the environment itself [Ref. 27: p. 31]. Peters says that although requirments may have been very fixed at the beginning, they tend to change and evolve with time. If for no other reason, the user's perception of the problem changes as does the designer's perception of that problem [Ref. 23: p. 70].

Change is inevitable during software design, and yet "planning for change" has long been given lip-service, at best. Neumann believes that planning for change is slowly being recognized as an important end in itself--and one that usually cannot be achieved by retrofits into an inflexible design [Ref. 28].

B. DESIGN IS SATISFICING

Most computer system developers will immediately argue this point. Developers of military systems would argue the longest and hardest. Why should the idea of satisficing be so controversial? Perhaps the answer lies in the past, when machine time was expensive and computer memory limited. These limitations do not exist at the same level today. In fact, satisficing occurs all the time. Conn states that the requirements for state-of-the-art systems are often scaled

down to respond to the need to cut the overall expense of the project or to meet time limitations [Ref. 26: p. 403]. Designers are, or should be, constantly aware of the trade-offs that are made in systems development, especially the classic trade-off, cost versus performance.

Several authors point out that a user should, in fact must, sacrifice an optimum design for a design which can cope at a satisfactory level [Ref. 29, 30]. John Munsun has been quoted as saying:

> Users must look at the economics involved in automation as a software-productivity solution. If a user can buy a payroll program that is almost what he needs for $10,000 or one that exactly fits his needs for $1 million, he must look at the trade-offs and reduce his expectations. [Ref. 30 : p. 66]

Satisficing has to do with more than economics. Lawrence Peters has said that the trade-offs for execution efficiency and ease of change must be evaluated and a compromise made. [Ref. 30]. Lockett emphasizes the role of user satisfaction when evaluating trade-offs. For her, user satisfaction is not based solely on the functional capability of a system, but on useability, reliability, and performance as well. Often the user cannot have everything (for example, both performance and functional capability) he or she wants in a system. The final product may be the result of compromise. Certain functional capabilities may be eliminated to achieve specific performance goals or, on the other hand, the user may be willing to sacrifice performance to obtain some functional capability [Ref. 31 : p. 157].

Several other authors emphasize the role of agreement, concensus, and negotiation [Ref. 32, 39, 33]. These authors contend that as system design progresses, alternatives are proposed and evaluated. The exact definition of a system

may not be as important as the concensus on the inexact definition which is attained. An example from Land serves to illustrate the importance of satisficing in software design:

> . . . . the designer has to be aware that building flexibility into systems can also be expensive, both in terms of design effort and operational performance. The designer is involved in a trade-off between the extra development and operational costs of designing a system which is adaptable and flexible--a very general system--or of designing a very specific system dedicated to the needs existing at the time of implementation, but which may be incapable of modification, and may have to be replaced if requirements change. [Ref. 29 : p. 67]

Satisficing may also involve psychological trade-offs as well as technical trade-offs. Madnick and Donovan relate an instance where two possible algorithms could have been used. The inefficient algorithm was chosen because the designer could not stand the suspense of waiting [Ref. 22: p. 491].

C. SOFTWARE DESIGN IS A WICKED PROBLEM

Horst Rittel has suggested that design problems are wicked problems [Ref. 13, 9]. These problems are ill-formulated, have confusing information, have many clients and decision-makers with conflicting values, and have ramifications in the whole system which are thoroughly confusing. Peters and Tripp have suggested that software design is a wicked problem. They believed that a comparison of the attributes and problems associated with software design and the characteristics of wicked problems make it apparent that software design is itself a wicked problem [Ref. 37]. A review of the properties of wicked problems and their relation to software design should help to put this notion in perspective.

Wicked problems have no definitive formulation. Any time a formulation is made, additional questions can be asked and more information can be requested. Our inability to define system requirements completely and unambiguously is a symptom of this problem. Current efforts in software development seem to be aimed at the symptom rather than the problem.

Several authors raise the possibility that a complete set of requirements is impossible, that a state-of-the-art system is almost by definition one for which there remains some degree of uncertainty at the time requirements are prepared. Under these conditions, it is hard to imagine a set of "complete" requirements, since the knowledge of the eventual system at that point can only be incomplete [Ref. 26 : p. 403].

Wicked problems have no stopping rule. Any time a solution is formulated, it can be improved or worked on more. One stops only because one has _run out of_ resouces, patience, or something else. Few would argue that there are clear stopping rules for software design. (Else why are there innumerable examples of cost and schedule overruns?)

Solutions to wicked problems cannot be correct or false. They can only be good or bad. This notion can be quite controversial among computer scientists. Granted, a computer system must work properly, especially in life-critical or life-threatening circumstances (hospital equipment or nuclear reactors, for example). But "work properly" has different meanings to different people, or groups of people, just as do "correct" or "true".[3]

----

[3]Mortimer J. Adler discusses the idea of "truth", an idea we judge by, in _Six Great Ideas_, Macmillan Publishing Co., Inc., New York, 1982.

Perhaps "good" and "bad" are poor choices as well, yet most of us readily acknowledge the differences when presented with "good or bad, for whom?" The distinction could be thought of in terms of 'technical success' and 'psycholcgical success'. Technical success is the degree to which the actual performance of the system matches its specification, while psychological success is the degree to which the end user has confidence in the final system [Ref. 36].

Another distinction can be made from the observer's point of view of a system: a system exists and is defined by the person(s) observing it. It is as acceptable, perhaps even laudable, as the observer perceives it to be. If a system works in the eyes of those who use it, then to those users that system is a good one. Conversely, if a system is observed as not working by those same users, then it is not good regardless of any other attribute it may have. [Ref. 33].

In solving wicked problems there is no exhaustive list of admissable operations. Any conceivable plan, strategy, or act is permissable in finding a solution and none can be prescribed as mandatory. Anyone in the profession can see that this certainly applies to software design (granted, there are at present a finite number of "design methodologies", yet each year this number continues to increase). The literature is replete with references to design methodologies: object-oriented design, data-oriented design, design based on finite-state machines, and so on.
See Table I for a large, and certainly incomplete, list of design methodologies.

Not only are we faced with many alternatives for a design "methodology", but we also are faced with innumerable alternatives for solving the subproblems in the particular design case at hand. There may be more than one way in

TABLE I

Design Methodologies

Mnemonic	Full Name of Methodology
ACM/PCM	Active and Passive Component Modelling
DADES	Data Oriented Design
DSSAD	Data Structured Systems Analysis and Design
DSSD	Data Structured Systems Development
EDM	Evolutionary Design Methodology
GEIS	Gradual Evolution of Information Systems
HOS	Higher Order Software
IBMFSD-SEP	Adaptation of IBM Federal Systems Division Software Engineering Practices
IESM	Information Engineering Specification Method
ISAC	Information Systems Work and Analysis of Changes
JSD	Jackson System Development
NIAM	Nijssen's Information Analysis Method
SADT	Structured Analysis & Design Technique
SARA	System ARchitect's Apprentice
SD	System Developer
SA-SD	Structured Analysis and Structured Design
SDM	System Development Methodology
SEPN	Software Engineering Procedures Notebook
SREM	Software Requirements Engineering Methodology
STRADIS	STRuctured Analysis, Design and Implementation of Information Systems
USE	User Software Engineering

which a target system development process can proceed simply because there are alternatie approaches available at the time the requirements are written. A decision between these alternatives may not be possible [Ref. 26 : p. 403].

For every wicked problem there is always more than one possible explanation. The selection of an explanation depends on the perspective, or world-view, used. The explanation also determines the solution to the problem. (For example, the high cost of software is often attributed to labor-intensive design and programming; poor requirements definition is often blamed for software "failures".)

No wicked problem and no solution to it has a definitive test. In other words, any time any test is "successfully" passed it is still possible that the solution will fail in some other respect. This characteristic of wicked problems is tied very closely to the idea of satisficing. If computer systems are built to be flexible, their design must be generalized. The aspect of flexibility is gained at the expense of efficiency (not that this is bad!). So, the system "passes" the test for flexibility but is very inefficient.

Each wicked problem is a "one shot" operation. There is no room for trial and error, and there is no possibility for experimentation. Many large-scale computer systems have this characteristic. In fact, software development is sometimes compared to building a bridge--once it is built there is no going back to the beginning to redesign and rebuild it (for any number of reasons).

Every wicked problem is unique. No two problems are exactly alike and no two solutions or strategies leading to solution can readily be copied for the next problem. This characteristic is very evident in software design. Military systems, for example, are certainly unique. Commercial or industrial problems are no less unique. Each organization has a unique structure, set of goals and objectives, set of interactions with the environment, cast of people, and set of needs.*

The wicked problem solver has no right to be wrong -- he/she is fully responsible for his/her action. There has been a growing skepticism among users regarding the abilities of software designers. Users have every reason to believe that the software designer "knows" the job.

---

*Note that what is being discussed is the overall problem, not a subproblem. The questions about reuseability and software components should be directed ONLY to subproblems, for obvious reasons.

Clearly, the designer must be aware of many of the factors which could affect the design. The designer must also be aware of the effects of design decisions. Allowances will and can be made for unusual unforeseen difficulties. But to hide behind the "This system meets the specifications you approved and signed" statement is going (and has gone) too far.

D. COMMUNICATIONS BETWEEN THE DESIGNER AND THE END USER

Perhaps the single, most widely noted problem area in software design is the problem of communication between the user and the designer. The recent literature emphasizes the need for extensive communications [Ref. 25, 29, 30, 35, 39, and 40]. The most common reason given for the problem is that users and designers speak with different vocabularies and find it difficult to completely understand each other.

Much of the literature which cites the need for closer communication is based on empirical and ancedotal reports. King and Rodriquez [Ref. 41], however, report an assessment of participation (and communication) in system development in an experimental context. The experiment tested four specific hypotheses (see Table II) about participative design which were stated in null form.[5]

The experimental results (see table III) indicate that participative design makes a difference, especially when viewing the "worth of the system".

---

[5]This only means that the 'claim', i.e., "accepted wisdom" in systems design, was set up as the <u>alternative</u> to the hypothesis, in accord with traditional hypothesis testing.

TABLE II

Hypotheses Tested in the Experiment

H1: Participation in the development of the system has no effect on the user's perception of the worth of the system.

H2: Participation in the development of the system has no effect on the amount of use which is made of the system when the user is faced with strategic issues for which the system was designed to provide support.

H3: The substantive inputs provided by participants in the design process will not be reflected in their usage of the system.

H4: The decision performance of participants in the design process will not be different from that of non-participants.

TABLE III

Results of the Experiment

H1: The null hypothesis is rejected.
This result indicates that managers who are involved in the development effort tend to perceive the system to be more worthwhile than managers who are merely given a pre-designed system to which they had no input.

H2: Cannot reject the null hypothesis.
conclude that the use of the system in terms of number of queries is not significantly different for design participants and non-participants.

H3: The null hypothesis was rejected.
it indicates that the substantive inputs provided by the participant group in the design and development phase of the information system are reflected in their actual use of the system.

H4: Cannot reject the null hypothesis.

As King and Rodriquez put it, the

> . . . . experiment provides <u>some</u> support for "participative design theory": (a) The inputs provided by participants appear to have been made use of in their use of the system, and (b) some positive attitudinal impact--in terms of systems "worth"--seems to be achieved through participation. [Ref. 41]

The experiment seems to confirm some deeply held convictions that participation in, and responsibility for, design implementation can result in elimination or reduction of communication problems [Ref. 29: p. 65].

There may be some reason to believe that the real problem with communication is not whether it takes place but whether the media of communication is appropriate. The fact that the designer has produced a comprehensive specification and that the user has 'signed off' the specification after due study, is not a guarantee that the designer has understood the user's needs, or the user the designer's specification [Ref. 29 : p. 65]. Stucki has suggested that charts, graphics, color pictures, and other aids should be used to enhance communications between users and designers; verbal descriptions alone are just as inadequate for describing software as they are for an architect building a house. [Ref. 30]. So, although communications may be a significant problem, its form may be equally as important.

E. SOFTWARE DESIGN IS LEARNING

Software design <u>is</u> learning, just ask any experienced program manager. They want someone with design experience

to head the design team [Ref. 46]. Without explicitly acknowledging it, these managers place value in the experience _learned_ from previous work. This "learning from doing" also takes place _during_ the design of a system:

> The reason for the discovery aspects of software design is the designer's learning curve. As the system is studied, analyzed, and a design formulated, certain features are recognized as needing attention while others are overlooked. As it becomes apparent which features are lacking, priorities shift. [Ref. 37]

If we accept that learning is an element of design, just how important is learning to design? In an experiment, Alavi and Henderson [Ref. 55] evaluated two strategies for systems development: evolutionary and traditional. By their definition, the evolutionary strategy emphasized the role of individual learning. They reported that the findings support the hypothesis that an evolutionary implementation strategy is more effective than a traditional strategy [Ref. 55].

They try to explain their findings this way:

> A model which offers an explanation for the findings is Kolb's experimental learning model [see Figure 3.1]. Kolb suggests that for a learner to be effective he/she must have the ability to engage in four types of activities: (1) involvement in new, concrete experiences, (2) observation and reflection of these experiences, (3) creation of concepts that integrate these observations into theories, and (4) usage of these theories to make decisions and solve problems. . . . The evolutionary strategy maps directly with a starting point at concrete experiences. In contrast, the traditional approach began with the development of a theory. . . . An explanation of the findings may rest in the support that the evolutionary strategy had for the learning process. [Ref. 55]

This model has some important implications for software design. For example, the perspective or world-view that the designers (and users) bring to a project become important (after all, we are starting from concrete

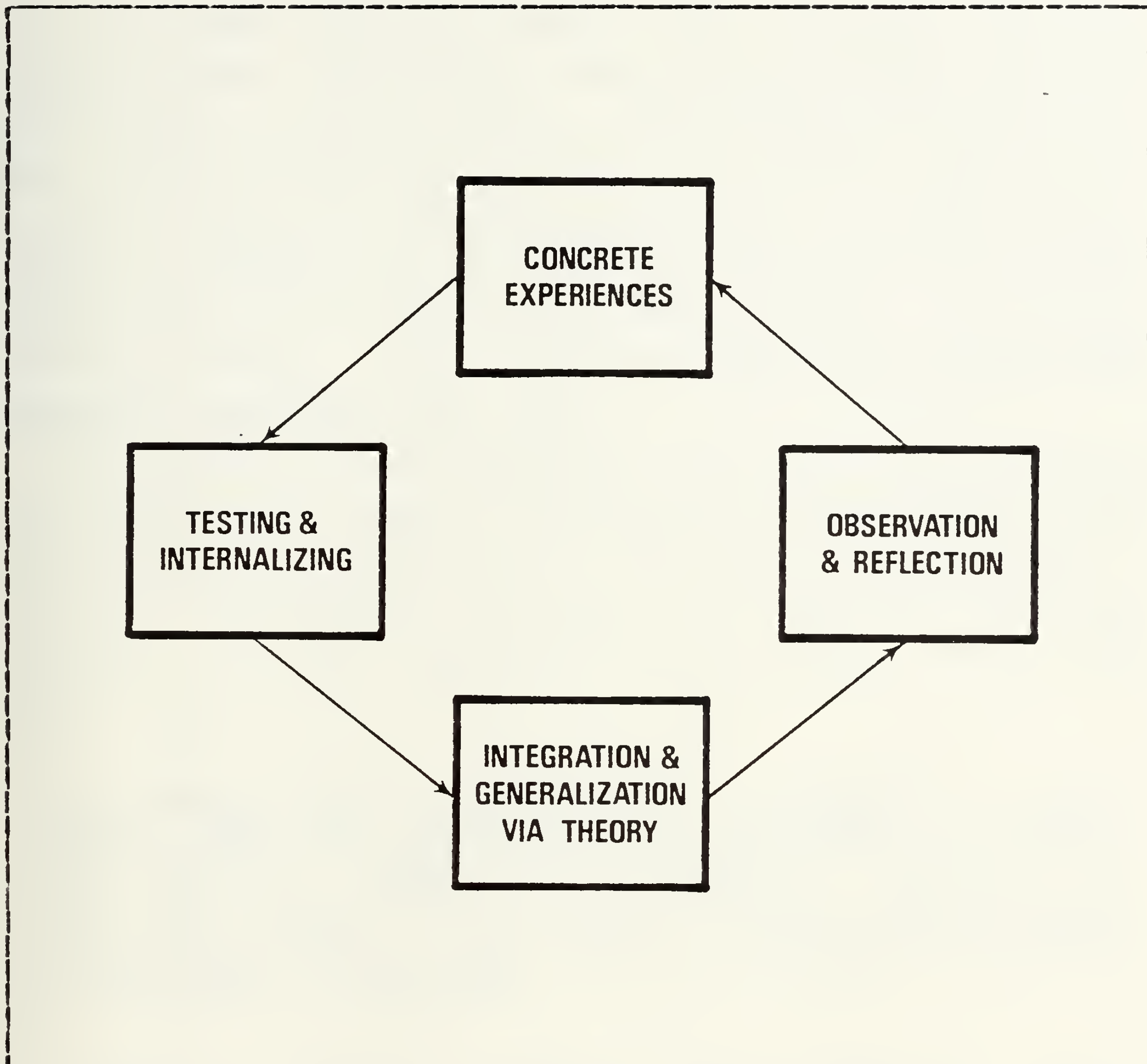

Figure 3.1 Kolb's Learning Cycle Model.

experiences). Greenspan and others believe that the ability to efficiently design appropriate computer systems and enable them to evolve over their lifetime depends on the extent to which real world knowledge can be captured [Ref. 43]. Wasserman [Ref. 35] takes the thought further by suggesting that members of the different groups concerned with design perceive the function of an information system differently. Misunderstandings of objectives can and do occur, many times leading to project failure.

Land [Ref. 29] also states that there are different ideologies and perspectives among the different interests involved in a systems study. Land suggests that managers meet this challenge by setting up a design team which contains representatives of all the major interest groups, making it possible for the different ideologies and perspectives of the participants to be made explicit, and for the different members of the group to <u>learn from each others different viewpoints</u> [Ref. 29].

How might the participation of users in the system design enhance or promote learning and real-world knowledge? Robey [Ref. 42] conducted an experiment that explored a model of constructive conflict in the MIS development process. His model (presented in Figure 3.2) is described here:

> User <u>participation</u> should lead to <u>conflicts</u>, which should then be satisfactorily <u>resolved</u>. However, conflict and its resolution are more likely to occur when users can exercise their <u>influence</u> in the development process. Conflict itself does not lead to its resolution; rather the increase in conflict makes resolution more difficult. It is only through participation and influence that conflict can be successfully resolved in this model. [Ref. 42]

There is other research which supports Robey's "constructive conflict". Boland [Ref. 54] compared two different processes of interaction in system design:

1. traditional--the designer conducts a traditional interview of the user
2. alternative--the designer and user share ideas, present mutual suggestions, and critique their suggestions.

His results are significant:

1. The alternative process produced higher quality designs with important implementation advantages.

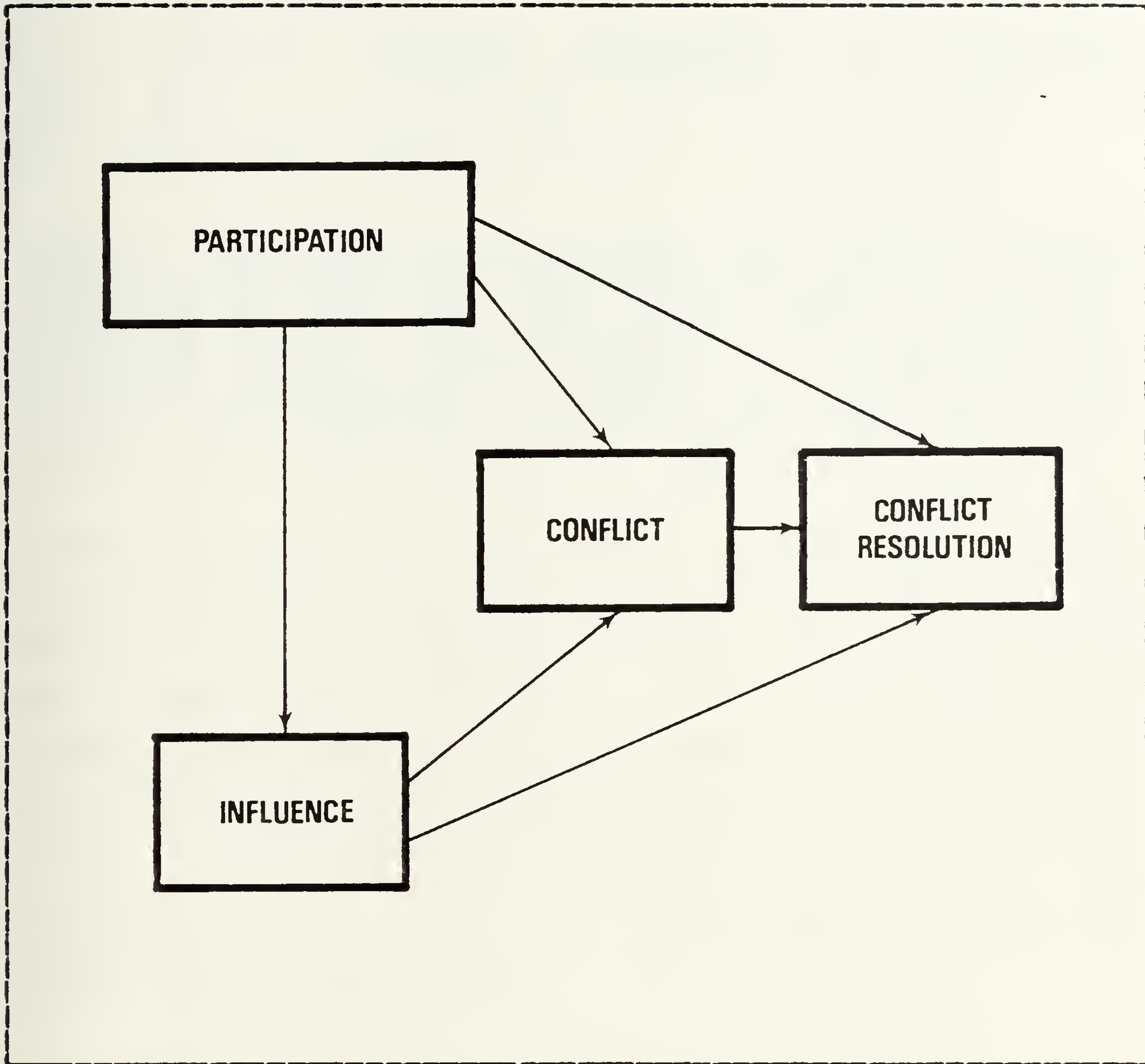

Figure 3.2 A Constructive Conflict Model for User Involvement.

2. The two processes produced designs which used different organizational control strategies.
3. Different processes may help to define different problems and thereby produce different, but equally rational, solutions. [Ref. 54]

Boland likens the problem solving process to a dance during which the designer punctuates his interaction with the user

in a series of teaching, suggesting, and critiquing.[6] Boland asks us to accept the notion of learning and the importance of real world knowledge:

> Let us accept that the viewpoint and implicit models held by designers will color their collection and interpretation of data about the needs of the organization they are designing for. This study suggests that understanding how that viewpoint builds a coherent design statement requires an understanding of how the designer interacts and exhanges information with his client. The interaction protocols may then be seen as mediating the process of completing the designer's "point of view" (creating the design statement). [Ref. 54: p. 896]

Robey's experiment lends support to Boland's findings: "It appears that participation does lead to perceived influence in . . . system development" [Ref. 42]. Robey's findings suggest that influence is used constructively to resolve conflict and that users learn how to exert influence towards conflict resolution as well as conflict generation as the development process proceeds [Ref. 42 : p. 82].

As we have seen, there is support that learning, argumentation, and a designer's world-view are important elements in software design.

F. SOFTWARE DESIGN HAS AN ORGANIZATIONAL CONTEXT

At first glance, the casual reader is apt to say "You are stating the obvious." Yet much of the current work in software design ignores the obvious. Land provides some evidence for this:

1. Users are uncertain about the affect the final system will have on their individual roles in the organization and on them personally.

----

[6]Compare Boland's "dance" and Robey's "constructive conflict" for software design to Rittel's "argumentation" in design (Chapter II).

2. The observation that the user operates within formal systems and that the formal procedure of the existing systems have been overtaken by less formal (but often more effective) unauthorized procedures.
3. The fact that those who are involved in the analysis process--DP specialists and users--are often not aware of strategic decisions made by senior management which could have an important bearing on the workability of the system.
4. New systems almost certainly include innovations; users and analyst/designers cannot predict managers' responses to innovations. Conjectures about people's behavior are no substitute for knowledge, and in innovation, such knowledge is not ordinarily available. [Ref. 29: p. 64]

Although Land cited these points as reasons for communications problems, they can equally serve as indictments against current software design. That is, <u>organizational</u> aspects of software design are often ignored.

Wasserman points out that organizations and computing environments are highly dynamic and that information systems must be designed for a changing organization [Ref. 35]. Chafin states that as computer systems become more deeply involved in the operations of organizations, they have larger social effects on these organizations. A new computer system may change the organization structure, the power structure, or the overall information flow structure in an organization [Ref. 40].

Zmud and Cox recognized the organizational aspects of software design in their discussion of a "change" approach to design and implementation:

> The change approach to MIS implementation strives to create an environment in which change will be accepted through the active involvement of affected organizational members, an intensive educational program, and, most importantly, the assigning of project responsibility to the MIS user. Additionally, a sense of mutual trust and committment must develop between participants so that a free exchange of beliefs and opinions is possible. [Ref. 53 : p. 37]

Zmud and Cox make no reference to wicked problems, yet their change process is recommended when (1) the organizational activity involved is ill-defined, (2) the MIS must interface with other organizational systems, and (3) substantial organizational change is expected. Compare these characteristics to Horst Rittel's characteristics of wicked problems (Chapter II).

Although there are several articles and references to organizational aspects of software design, two authors stand out. Kling and Scacchi have written two extensive articles, [Ref. 59 and 60], which stress the need for an awareness of and attention to organizational and social aspects of system design. Their latest work [Ref. 60], develops a family of models (called web models) which they believe helps to "make better predictions of the outcomes of using socially complex computing developments". These models are contrasted to 'discrete-entity'--rational and traditional--models. Their work attempts to abstract a set of principles, characteristic of web models, from analyses published in the literature.

Kling and Scacchi stress the importance of perspective in the "social analyses of computing". They identify six perspectives, four of which predominate:

1. Formal-rational
2. Structural
3. Interactionist
4. Political

Their point in discussing these perspectives is that each "casts a different light" on the significant aspects of the design problem.[7]

Further discussion of the work of Kling and Scacchi is beyond the scope of this work. The point to be made of their work is that software design is conducted in an organizational framework:

> In contrast to the discrete-entity models, which gain simplicity by ignoring the social context of computing developments, web models make explicit the salient conections between a focal technology and its social and political contexts. [Ref. 60 : p. 3]

G. SOFTWARE DESIGN IS EVOLUTIONARY

Much of the current practice in software design is constrained by a model popularly termed the 'waterfall' model. Tom Gilb aptly sums up the attitudes of most software professionals:

> It seems that they recognize, as yet, only one type of life cycle. In particular, they seem to be speaking of a revolutionary life cycle (like the birth of a human) as opposed to a more evolutionary life cycle (such as the development of the human species). [Ref. 34]

---

[7] Kling and Scacchi present an extensive discussion of the social dynamics of system design in [Ref. 59]. Their discussion is based on the four perspectives mentioned as well as two others: human relations and class politics.

Other authors also complain about the current life cycle model. Brittan is concerned that the serial definition of the project development cycle, known as the linear strategy, embodies one fundamental concept: that an activity follows logically from its predecessor so that each stage is complete before the next begins [Ref. 36]. McCracken and Jackson seem to be the most critical of the current life cycle model. They believe that any form of life-cycle is a project management structure imposed on system development. Furthermore, they point out that the current life cycle model is either a very much simplified model (which is worthless) or unrealistic [Ref. 27]. Podolsky [Ref. 24] argues that the current model (which he terms 'Classic Development') is "very, very good" when it is successful, but that when it fails, "it's horrid". He attributes the success and failure of Classic Development to the type of problem which will be solved: classic development is good for well-defined, highly structured, change-resistant problems; it fails when presented with an ill-defined problem, changing participants, and changing requirements. Zvegintzov [Ref. 57] has two objections to the current life cycle model. First, it does not portray a systems life, only the creation, development, or youth of a system. It does not include adulthood and is vague about operation and maintenance. Second, it is not a cycle, it portrays a linear path and does not, as a cycle must, return to its beginning [Ref. 57]. Gladden even goes so far to say that the software life cycle may be harmful to the software profession. See Figure 3.3 for Gladden's representation.

These arguments, and others, begin to raise a question about the validity of the linear strategy. The linear strategy places a great deal of reliance on the studies and efforts made in the earlier 'stages' of software development. Yet this strategy ignores the fundamental aspects of

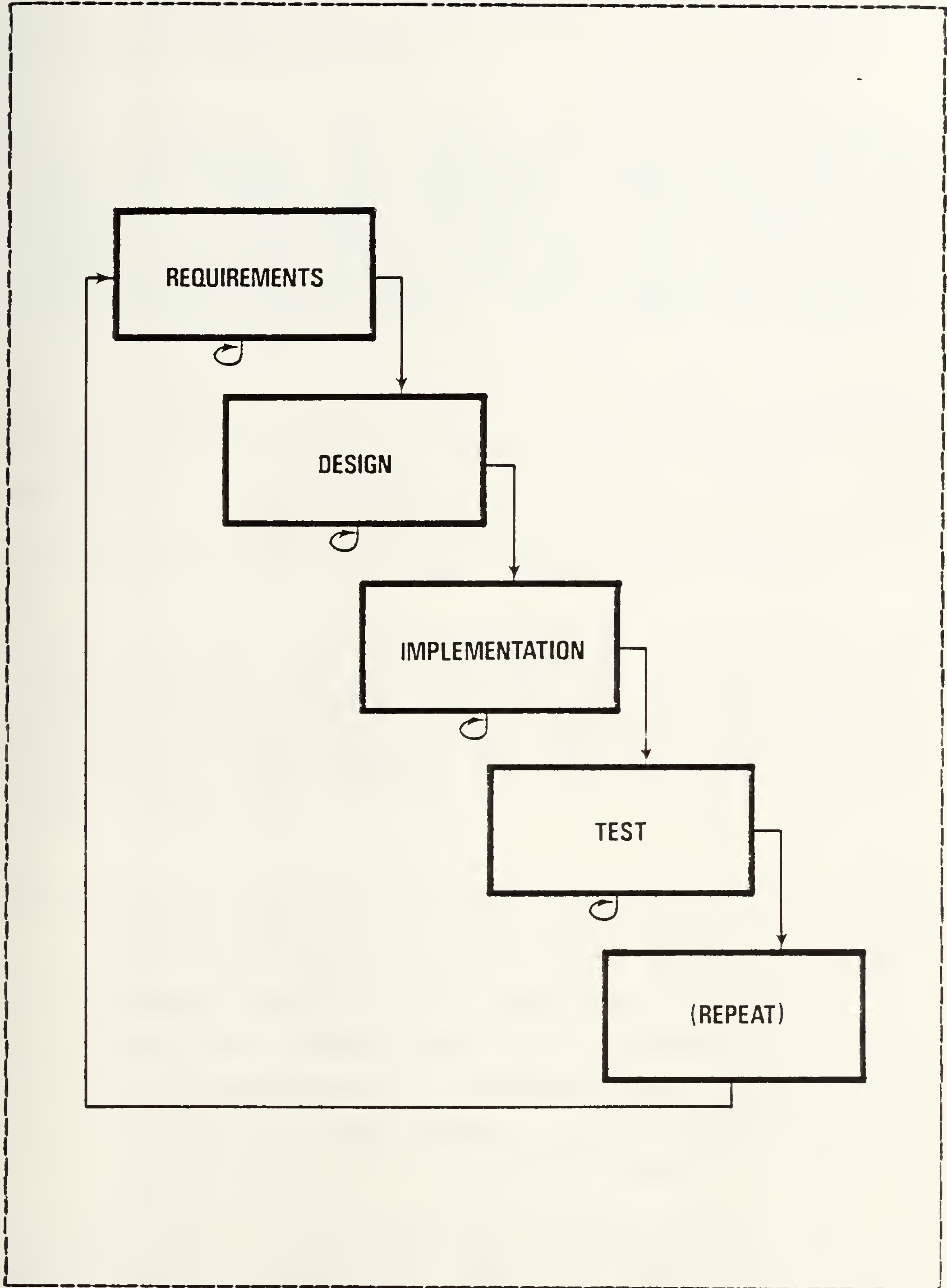

Figure 3.3 Typical Life Cycle Representation.

design described in Chapter II. Brittan places this predicament in perspective:

> In a majority of cases, particularly when the organization responsible for designing and implementing the system has experience of similar systems and when the users are clear about what they want, the linear strategy is pefectly satisfactory and produces good results. Too often, a project starts on the linear strategy but the initial requirement is vague, overambiticus or fails to meet the real need: in fact the requirement is still fluid. The project then proceeds in a series of short locps as the requirement solidifies. . . . [Ref. 36]

Now it becomes clear why Gladden's representation in Figure 3.3 appears as it does. To make up for the reality of software design, the practice is to use a 'loopy linear' strategy. That is, to proceed in a series of relatively haphazard and short-term locps. Again from Brittan:

> Some loops are inevitable. One of the symptoms of excessive loopiness is a feeling of antipathy between the different groups associated with the project, particularly if they are geographically dispersed (the we/they syndrome);the system designers will grumble about users never knowing what they want and users will be annoyed by the apparent lack of good project management as the system cverruns its budget in both time and cost. [Ref. 36]

Brittan gives other reasons why the linear strategy is poor:

1. when analysts refine the requirements of a system, their investigations and studies frequently throw up problems which were not suspected at the outset.
2. the linear strategy can only be based on studies and investigations made by analysts; users, who determine the success of the system, are not usually adept at the conjecture and extrapolation needed to understand those studies.

Land [Ref. 29], Brooks [Ref. 46], Podolsky [Ref. 24], Zave [Ref. 32], and Lehman [Ref. 47], to name a few, have all argued that a system will require substantial, continuing

changes after the client begins to use the system. We tend to relegate this phenomenon to 'Maintenance'. But this isn't enough. Consider this comment by Land:

> The conventional model of the systems life cycle assumes that an analysis and feasibility stage precedes the detailed design stage and that this will be followed by a specificiation and agreement of the specification for the system. At that point the design of the system is often frozen. For a typical information system the stages preceding the design freeze take between 20% and 35% of the total time required for the develoment of the system. For between 65% and 80% of this time the design of the system is not to be modified, even though the "world" is changing all the time. In practice, even a frozen design gets modified if the system is seen to be becoming irrelevant to real requirements. Further, inconsistencies in design are discovered during the construction phase as a result of "systems queries". [Ref. 29 : p. 68]

Software design, no matter how hard we try otherwise, is simply not linear. The literature clearly supports an evolutionary strategy, yet our practice has not recognized this.

H. SUMMARY

The preceeding discussion shows that there is support in the literature for reassessing our view of software design. Software design is symmetrical, but we currently do little to recognize that symmetry. Software design is satisficing, yet there is constant emphasis on optimization, often for its own sake and forsaking approaches that enhance the useability or quality of the software. Perhaps, without consciously noting it, we are also concerned with the "best" design and dooming the project to mediocrity, at best, and perhaps catastrophe.

Software design, especially for large-scale systems, is certainly a "wicked problem." All the evidence is there; it only remains to acknowledge that fact. We are well aware that communications between the designer and user are

all-important. Yet, we have not really given much thought to the *medium* of exchange. Software design is a learning experience. Designers learn that projects are more complex than expected and users learn never to trust designers. This may be a harsh critique, but the point is well illustrated: all parties gain something from the experience of software design. Let us recognize the worth of this.

The organizational context of software design has long been ignored, particularly in military systems. We must not forget that the computers are *to help the people in a system to perform well*, not to control the people as a part of the system. Finally, we are beginning to recognize that software design is evolutionary. There really is no "end" to a project, simply a restatement of the goals originally identified.

Although seven characteristics have been stated and discussed, their interdependencies are obvious. None of these characteristics is mutually exclusive of another. Rather, each builds on the other. Although there are innumerable implications in that statement, the remainder of this work will examine one approach which may help us to consider the seven characteristics of design in software design.

IV. THE SOFTWARE PROTOTYPE

A. INTRODUCTION

For the last 35 years, systems software development has been based on the so-called 'system development cycle.' As shown in the last chapter, there are several arguments against such a cycle. Perhaps the most telling argument lies in our process controls. Several authors [Ref. 61 , 62] have pointed out that in response to uncertainty and increased complexity, there is a tendency to define and structure (and increase!) management controls. Correspondingly, precise requirements definitions have been emphasized. Berrisford and Wetherbe [Ref. 61] believe that there is a major conceptual flaw in the traditional view of systems development. This is that system design assumes that management knows what information is needed and it is difficult, if not unrealistic, to ask managers to define their information requirements on paper.

How do software designers cope with this problem? Rich and Waters [Ref. 63] have explored this question and theorize that software designers cope with complex design problems by using several mental tools, one of which involves simplifying assumptions. The use of simplifying assumptions is both necessary and commonly used when constructing large and complex systems:

> Given a complex programming problem, expert programmers typically choose simplifying assumptions which, though false, allow them to arrive rapidly at a program which addresses the important features of the problem without being distracted by all of its details. The simplifying assumptions are then incrementally retracted with corresponding modifications to the initial program. Often the main questions can be answered using only the initial program. [Ref. 63 : p. 150]

This use of simplifying assumptions in software design is very much like the idea of the tentative solution, which was introduced in Chapter II. Such a tentative solution is only a simplified system. Earl [Ref. 64] calls these simplified systems prototypes. Carrying this one step farther, Naumann and Jenkins define a prototype system as "a system that captures the essential features of a later system." [Ref. 62]. The sections which follow will describe the prototype process, the role of prototypes as models, the ways in which prototypes are used and concludes by showing how the set of seven desing elements are supported by software prototypes.

B. THE PROTOTYPE PROCESS

The terms protctype and prototype systems have become rather common lately, found in both the management literature (Harvard Business Review, for example) and the software engineering literature (proceedings of conferences and workshops especially). Although the term prototype has become standard, early descriptions of the process were called "heuristic development" and "iterative enhancement" [Ref. 61, 65].

Regardless of how each of us may use the term, there is general agreement that the main purpose of prototype systems is exploration and experimentation; "the aim of the early prototype is to learn, to find out, to discover." [Ref. 68, 64, 66]. In keeping with their purpose, prototypes are relatively inexpensive, flexible, and simplified systems. Bally, Brittan, and Wagner describe the prototype process:

> In the prototype strategy, an initial and usually highly simplified prototype verson of the system is designed, implemented, tested and brought into operation. Based on the experience gained in the operation of the first prototype, a revised requirment is established, and a second prototype designed and implemented. The cycle is repeated as often as is necessary to achieve a

> satisfactory operational system, bearing in mind the possibly escalating cost of each subsequent cycle; it may well be that only one prototype is necessary before producing the final system. [Ref. 68: p. 23]

From this description, four steps are evident [Ref. 62]:

1. Identify the user's basic information requirements.
2. Develop a working prototype.
3. Implement and use the prototype.
4. Revise and enhance the prototype.

Figure 4.1 illustrates the prototype process.

A prototype system must be implemented quickly, perhaps in hours or days, certainly no more than two or three weeks. The advantage here is in the user-designer interactions: the user is given a working system to operate and criticize, the designer receives responses based on the user's experiences. The quick response of the designer guarantees that the first prototype will be incomplete. This aspect is important: there is an explicit understanding between the user and designer that the system will be incomplete, that a prototype is <u>meant</u> to be modified, expanded, supplemented, or supplanted [Ref. 62].

C. PROTOTYPES AS MODELS

Many authors consider prototypes to be models [Ref. 64, 82, 69]. As models, prototypes reduce risk and test alternative designs through live operation. [Ref. 64].

Three aspects of prototypes as models are important. First, models are abstract:

> The critical skill of system design is . . . claimed to be explication of the implicit models in managers' minds, of their decision-making processes and views of their organisation and environment. [Ref. 64 : p. 163]

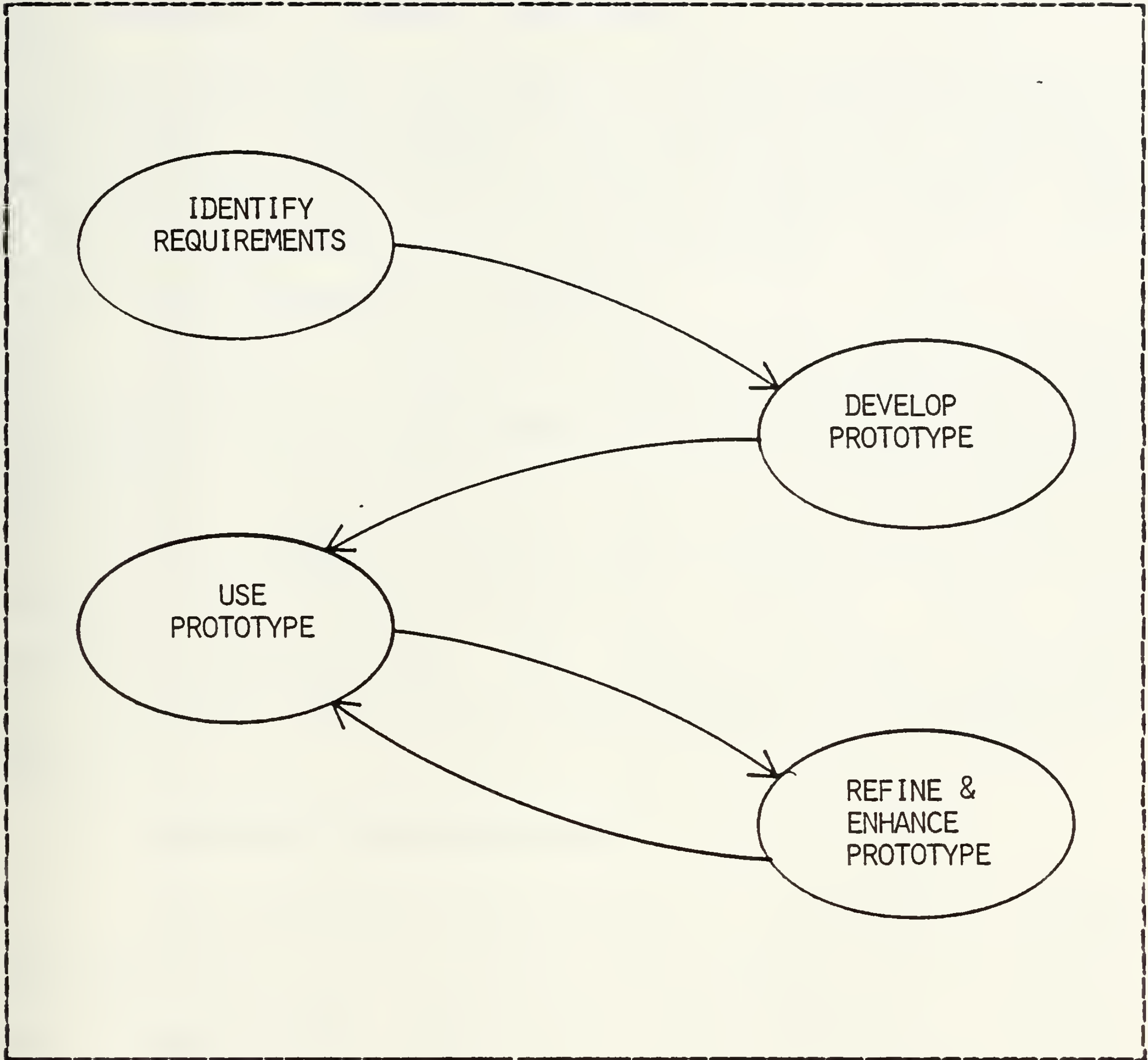

Figure 4.1 The Prototype Model.

Second, managers prefer simple models at first. As they begin to understand the models, they become involved with the design and implementation to build more realistic systems. [Ref. 64].

Third, a prototype is subject to modelling effects. That is, as a model, the prototype is only a limited version of the final system. So, a prototype is one kind of scale model, accurate in some ways, inaccurate in others [Ref. 69].

D. STRATEGIES TO PRODUCE PROTOTYPES

Three strategies are generally recognized for producing prototypes, 1) methodologies (in current use), 2) executable specifications (state-of-the-art and research issues), and 3) automatic programming (a research topic).

1. The 'Methodolcgy' Strategy

There are three basic methodologies which are used to produce software prototypes. First, in screen and report formatting, the designer produces a set of user interfaces which will be similar to the final system. Second, in partial and incomplete implementation, the designer and user identify only a subset of the total problem. Third, for selective implementation, the designer develops components of the final system and then integrate the components. [Ref. 71]

2. Executable Specifications

The executable specification, the second technique for prototyping, is a current 'hot' topic in the computer science literature. Davis [Ref. 72] describes a software tool, the Feature Simulator, which "executes" formally written requirements specifications for real-time systems. Feather [Ref. 73] proposes a methodology for developing protctypes from specifications based on the transformation of "specification constructs" into an implementation. Perhaps the most ambitious work on executable specifications is that reported by Cohen and others. They believe that "a protctype serves to mitigate both imperfect communication and lack of forsight (sic)." [Ref. 74]

The solution Cohen and others have adopted separates the imperfect communication and lack of foresight issues by having a formal specification language which unambigiously

describes systems, and a separate tool (symbolic execution system) which helps the reader to understand any particular specification. This tool can be used by the specification writer to validate the specification and by the implementor (or buyer) to understand what exactly has been specified (i.e., how the pieces interact). "Given the specification and the tool, a prototype will not be needed." That is, if the designers can completely specify the requirements and they then use the symbolic execution system, Cohen and others believe that it is no longer useful to develop a prototype.

But consider the following comment by Taylor and Standish:

> . . . . having a precise specification language is of no help, since the user really doesn't know what statements to make in such a language-- that is, he can't articulate his needs if he doesn't know what they are regardless of whether or not there is a precise language for stating them. [Ref. 78 : p. 160]

Executable specifications clearly are controversial, especially when they concern prototypes. Whether such a technique gains prominence will depend on advances in software engineering tools.

3. Automatic Programming

Automatic programming is probably farther away, technically, than the executable specification. Automatic programming can be thought of as programs that help people write programs. The general goal of automatic programming is to allow the software designer to think of the problem abstractly, in a way which is natural and comfortable. Automatic programming systems are characterized by specification methods (formal, 'by example', or natural language), the target language (the language in which the system writes

the finished program), the problem area (area of intended application), and the method of operation (theorem-proving, program transformation, knowledge-engineering, or traditional problem solving) [Ref. 100]. One advantage of automatic programming is that it could allow for more informality than an executable specification language [Ref. 70].

E. USES OF PROTOTYPES

Generally, there are three uses of prototypes, 1) to clarify user requirements, 2) to verify the feasibility of a design, and 3) to create a final system. [Ref. 75].

1. To Clarify the User's Requirements

By far, the most popular use of prototypes is to clarify the user's requirements. McCracken [Ref. 67] believes that traditional written specifications do not bridge the communications gap between the designer and the user. He states that prototypes _encourage_ users to change their minds about what they want, until the system is _useful_.

To highlight the problems encountered in requirements documentation, Mason and Carey [Ref. 76] make a distinction among three types of documentation:

1. A textual list of requirements (the most commonly used)
2. An interpretive model (gaining in popularity, especially in military systems)
3. A working model--a prototype

The textual list, the traditional method of describing requirements, has a distinct disadvantage. There is a psychological distance between a textual list and what the users will eventually receive. A lengthy (often boring)

document does not easily convey a realistic sense of how the system will operate and suit the user's needs [Ref. 76].

Interpretive models include SADT and USE. These models use top-down decomposition to manage the complexity of large systems. The more detailed these tools are (or become), the more specialized the language used. This presents a significant learning burden to the user [Ref. 76].

Prototypes, on the other hand, present a more realistic view of the system to the users. The users can easily relate their experience with the prototype to their requirements.

2. To Verify the Feasibility of Design

When prototypes are used to verify the feasibility of a design, the designers and users are evaluating the internal design of the software [Ref. 75]. After the prototype is developed, several aspects of the design could be evaluated: the prototype could be used to implement and evaluate certain design decisions; the prototype could be used to develop and test a production system; the efficiency of the protctype could be examined; or the prototype could be developed on one machine, and the final system implemented on the target (or production) machine, when it becomes available.

3. To Create the Final System

Prototypes may be used to create the final system. This means that part or all of the final version of the protctype may become part of the production system [Ref. 75]. Examples of this technique might include database management system (DBMS) applications. For example, once created, the prototype might remain unchanged especially if the system efficiency is satisfactory. On the other hand, critical (or perhaps all) of the system would be

recoded for efficiency, either in the DBMS language, in a host language, or in assembly language.

F. PROTOTYPES ADDRESS THE ESSENTIAL DESIGN ELEMENTS

1. Prototyping is a Symmetrical and Adaptable Process

Prototypes explicitly address the symmetry and adaptation necessary in software design. Naumann and Jenkins [Ref. 62] believe that prototypes provide an appropriate response to changes in the development process (problems to solve and available resources) as well as to changes in the environment. Bally, Brittan, and Wagner state that the prototype strategy is an admission of failure, an admission that there will be circumstances when we will be unable to develop the right system on the first attempt [Ref. 68]. Earl's comment perhaps best expresses the overall idea of symmetry and adaptation:

> The prototype system . . . allows . . . design by discovery as much as by prediction, where the unexpected results may be as significant for design as the expected. (emphasis added) [Ref. 64 : p. 166].

2. Prototyping 'Tames' the Wicked Problem

In Chapter II wicked problems were described as problems where the information is confusing, where there are many clients with conflicting values, and where the ramifications in the whole system are thorougly confusing. Compare those characteristics to the experiences of Asner and King:

> . . . . the protctype approach works when users do not know their specific requirements, [where] the effectiveness of any on particular approach cannot be easily assessed without real-life experience, . . . [where] the system will be an integral part of the day-to-day activities of the users. . . . [Ref. 79 : p. 30]

Developing prototypes does more than recognize wicked problems. The designers and users of prototypes explicitly acknowledge such things as:

1. Wicked problems have no definitive solution--as Bally and others have stated, prototypes are an admission that more questions can be always asked and more information can be requested.
2. Every formulation of the wicked problem corresponds to the formulation of the solution (and vice versa)--there is an explicit understanding between the desinger and user about basic assumptions that will be made when designing a prototype, especially the first version; the protocype strategy is designed to cope with a fluid situation and fuzzy requirements [Ref. 68].
3. Wicked problems have no stopping rule--designers and users realize that prototypes may be continually modified or refined until some external limit (time, resources, production need, user satisfaction, etc.) is reached.
4. Solutions to wicked problems cannot be correct or false. They can only be good or bad--prototyping explicitly recognizes the notions of "technically" correct and "psychologically" correct. Users continually ask for refinements until they become satisfied (i.e., where the system is technically <u>and</u> psychologically correct).
5. In solving wicked problems there is no exhaustive list of admissable operations--prototypes allow designers and users the freedom to explore and experiment.
6. No wicked problem and no solution to it has a definitive test-- designers and users become quickly aware that prototypes clearly identify tradeoffs. The

prototype may be flexible and sacrifice (i.e., "fail" the test for) efficiency.

3. Software Prototyping is Satisficing

Recall from Chapter II Simon's argument that people accept alternatives which are good enough, not because they want to, but because they have no choice. In Chapter III evidence was presented which clearly shows that software designers constantly balance trade-offs and are forced to accept satisfactory alternatives, rather than an optimal alternative.

The process of developing a prototype explicitly deals with satisficing by recognizing the interaction among the user, designer, and system. Conflicting goals and priorities are inevitable. Negotiation between the designer and user will lead to a satisfactory system.

In the prototyping process, the designer constructs successive versions of the system, compromising and resolving conflicts between the context (that is, user needs and desires) and the form, as constrained by technology and economics [Ref. 62 : p. 37].

4. Prototyping is Communicating

The prototype facilitates communication between the designer and the user. The basic model of the protoype process shows that communication is a necessary element of the process. Without communication there is no prototype. Mason and Cary [Ref. 76] believe that prototyping overcomes the fundamental problems of communication between users and designers. Naumann and Jenkins [Ref. 62] emphasize the roles participants have and believe that prototyping stresses the interactions between the user and the designer.

Participation in software design can be painful [Ref. 64], yet

> Users play more active roles in prototyping than is possible with traditional development methods. Users set the development pace by the time they spend using and evaluating the prototype. They decide when the cycle of evaluation and refinement ends. [Ref. 62 : p. 37]

The prototype approach exploits the interaction between the designer and user. Contrast this with the carefully monitored interaction in the traditional approach.

5. The Software Prototype is a Learning Aid

Several authors [Ref. 64, 68, 66] agree that the very purpose of the prototype is to allow the user to learn about the system; experience with the system is the most valuable product. When prototyping, both designers and users learn, developing a system which is more realistic in its economic purpose, organizational context, and technical performance [Ref. 64 : p. 166].

Earl [Ref. 64] believes that prototype systems permit action learning and that there are few other vehicles available for live and flexible organizational development. As a vehicle for learning,

> . . . . the protoype model is the most effective representation possible since it enables evaluation of the proposed design in context. The prototype model is the representation that anticipates evaluation of the design in its operating environment. [Ref. 62 : p. 33]

6. <u>The Prototype Processs Accounts for Organizational Issues</u>

As pointed out in Chapter III, the organizaitonal context is an important consideration in systems and software design. Informal organizational structures and the sub-elements of organizations play large roles in the success or failure of a system. As an experiment, the prototype provides an opportunity to test the impact of a system and experiment on the organization's interfaces, at least reducing the risk of a nonviable system and also providing opportunities for introducing and monitoring job satisfaction improvements, organization development, and the like [Ref. 64: p. 164]. Earl believes that prototypes are relevant to organizations because of individual differences among people in the organization:

> . . . . the prototype methodology may be relevant, for different values, perceptions and perspectives do exist among different interest groups, but the different implications and impact of a system design may not be appreciated until it is implemented; indeed all the options may not be apparent. With a working prototype system design values may be explicated and stakeholders counter the technical thrust of the specialists. . . . [Ref. 64 : p. 165]

To say that prototyping "solves" the organizational issues in software design is, however, going too far. Prototyping deals <u>explicitly</u> with the issues, yet requires quite a bit of "orchestration". The management of the process is not without political consequences [Ref. 66].

How do we measure the worth of a prototype as it contributes to our design of software? Earl answers this question with the following statement:

> Possibly the most valuable contribution of the prototype methodology is to foster a climate of system appreciation, user creativity and experimentation, intelligent use and organisational learning. [Ref. 64 : p. 166]

7. <u>The Prototype Process is Evolutionary</u>

That the process of protoyping is incremental and evoluticnary shold ccme as no surprise. The important point is that the prototype process, again, explicitly deals with the issue. Software design has been shown to be evolutionary, yet traditional software development is unable to deal with it. Naumann and Jenkins [Ref. 62] state, as a 'principle', that "[p]rototyping represents and parallels the dynamic process of growth, change, and evolution existing in any living system."

A survey of the literature reveals an interesting pattern among the models for prototyping. Although most authors will agree that the traditional life cycle is not evolutionary, with the exception of Naumann and Jenkins [Ref. 62], (see also figure 4.1) Basili [Ref. 65], Bally and others [Ref. 68], Earl [Ref. 64], Mason and Carey [Ref. 76], and Zvegintzov [Ref. 57] <u>all</u> attempt to force a <u>cyclic</u> structure on software development.

Perhaps a review of evolution is in order. When some thing (animal, organizaiton, or design) evolves, it begins simply (a few cells, a few people, a few details and many simplifying assumptions) and grows in complexity, often changing remarkably from its humble beginnings. This process is clearly <u>not</u> cyclic. Rather, a better image is the spiral, much like the spiral coil of the shell of the Nautilus, growing in size yet maintaining the essential nature it began with.

Figure 4.2 illustrates the evolutionary nature of prototypes. Each "chamber" can be considered to be a single prototype, the wall of the "chamber" denoting the point of refinement and enhancement. The only restrictions on the number of "chambers" (prototypes) are in the environment (exhausted resources, end of time, too complex or unwieldy, and so on).

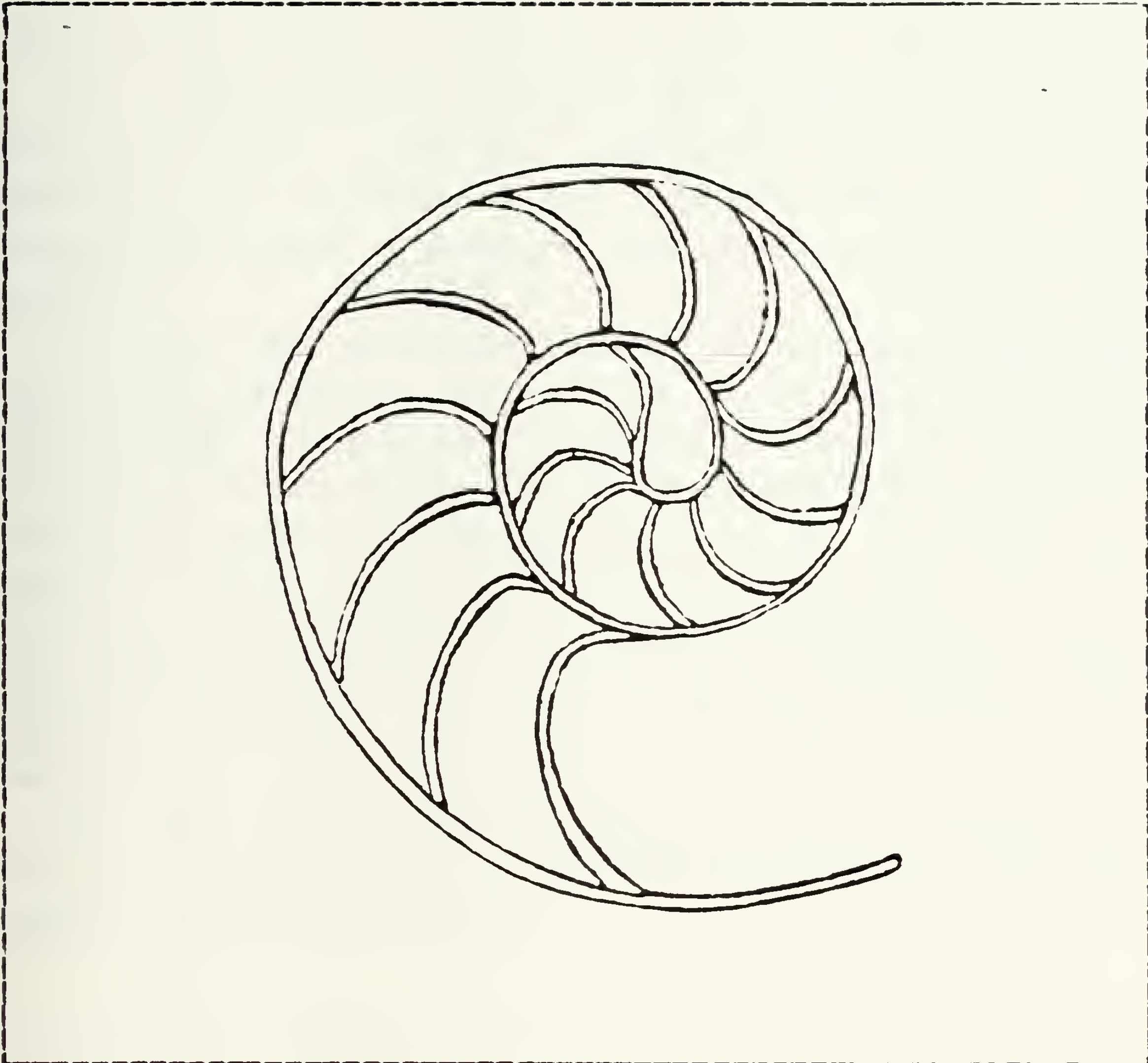

Figure 4.2 Evolution of Prototypes.

G. SUMMARY AND INTERMEDIATE CONCLUSIONS

This chapter has explored the multi-faceted aspect of the software prototype: the process, its role as a model, construction strategies, and uses. The chapter concludes with a persuasive argument that prototypes explicitly support the seven design elements.

Several conclusions can be stated at this time. First, the current practice of software engineering only recognizes a few of the design elements described in Chapter II. Software design completely ignores the fact that these elements are interrelated and mutually dependent. The traditional method of software development only worsens the problem.

Second, the prototype approach to software design and development naturally supports the set of design elements. For example, the prototype approach encourages, requires, and explcits the interaction and communication between the user and designer. By making this explicit, prototypes will lead to a better design.

Third, developing better systems, delivering them on time and within budgets are in our best interests. The prototype approach will allow software engineers and designers to achieve these goals.

The next chapter briefly describes software engineering environments and how such an environment could and should support software prototyping.

V. THE SOFTWARE ENGINEERING ENVIRONMENT

A. INTRODUCTION

Most authors agree that prototyping has become possible through recent developments in computer technology [Ref. 61, 62]. Collectively, this technology is called the software engineering environment (SE^2) [Ref. 83], the programming support environment [Ref. 107, 105], or the software development environment [Ref. 84].

There are as many definitions for, as there are references to, a software engineering environment. The definition offered by Hausen and Muellerburg seems to be the most satisfying:

> [A Software Engineering Environment is] an instrumented and organized software development laboratory where many people cooperate with each other in a fully organized working process, in the design, construction, examination, tuning and maintenance of software. [Ref. 83 : p. 147].

Generally speaking, the literature cites two approaches to computer-aided design for software development: 1) the SE^2 is a systematic approach, and 2) toolboxes or toolkits which support specific software development activities [Ref. 85]. The UNIX development environment is an excellent example of the toolkit approach [Ref. 86]. The facilities of UNIX may be thought of as a "tool kit" from which the developer can select tools that are appropriate for a specific task. Detailed discussions of the UNIX environment and available tools can be found in [Ref. 87 , 88].

The toolkit approach, however, has been criticized because:

1. Tools are not organized to support specific software development methodologies;
2. Tools do not capture management or control data for software development; and,
3. Individual tools are largely uncoordinated [Ref. 89].

Lauber has reviewed 11 tool systems in practical use and finds that only two systems (PSL/PSA and PDL) are in wide use.[8]

There are several "programming" environments in active use (for example, Interlisp [Ref. 90 , 86 , 91]), or planned (for example, the Ada programming support environment (APSE) [Ref. 93]). Unfortunately, there is no SE^2 which specifically supports the prototype process. This chapter will first describe some general characteristics of SE^2s and then explain those elements of SE^2s which are needed to support software design and prototyping.

B. CHARACTERISTICS OF SOFTWARE ENGINEERING ENVIRONMENTS

1. Development Support Tasks

There is general agreement that an SE^2 supports three development "tasks": 1) software production management, 2) technical aspects of software development, and 3) user participation in applications development [Ref. 83, 94 , 95]. An SE^2 aids software management by "capturing" information about design decisions and the progress of the development itself. An SE^2 supports software development by providing automated tools. During the development of specific applications, the SE^2 places special emphasis on the role of user-designer interaction.

----

[8] Problem Statement Language/Problem Statement Analyzer and Program Design Language. A detailed review of the various tools and environments in current use can be found in _Symposium on Software Engineering Environments_ , Huenke, editor.

2. Integrated

An integrated SE^2 will support the three development tasks by unifying the tasks into an ensemble. Integration applies to the ease of using and the ease of documenting those activities associated with individual tools [Ref. 84]. Perhaps one of the more important characteristics of an SE^2, integration makes it easier to combine various tools in order to perform a specific function.

3. Uniform

A variety of automated tools are used by the SE^2 to support the three development tasks. For reliable operation, the tools must be consistent with one another [Ref. 84, 94 , 95 , 96]. If one tool is consistent with the rest, the SE^2 will be easier to use. It is easier to learn and use special formats and command structures when they are consistent among all of the tools.

4. Support a Solution Strategy

The technical aspects of software development require the SE^2 to support two solution strategies, one general and the other specific. Generally, Soni and others believe that the SE^2 must support different ways of solving the problem. [Ref. 84]. That is, the SE^2 should support many different ways of solving problems. It should be flexible enough any problem-solving strategy. For the specific strategy, Wasserman and others believe that an SE^2 must support both the software life cycle model (the 'waterfall' model) and any particular software development methodology which does not diverge very much from that model [Ref. 94, 95, 97]. In either case, the objective is the same: to arrive at a solution.

5. Adaptable

For practical reasons, an SE^2 should be adaptable. In most organizations, each of the development tasks is covered by different organizational groups, each with their own styles, attitudes, and so on. Also, the individuals within each group bring different perspectives to the job. With such a wide range of personalities, a collection of tools should be flexible, changeable, even extensible [Ref. 84]. The SE^2 should be able to adapt to the designer's (or user's) sophistication and should provide defaults. Defaults could be easily changed as users become more sophisticated [Ref. 94, 95, 96].

6. Functionally Unique

Within each development task, there are a number of unique functions. To reduce ambiguity, misunderstanding, and errors, tools within an SE^2 must be functionally unique. That is, they must have a singular purpose [Ref. 84, 94, 95, 96]. Each tool must be limited to a single design function.

7. Interactive

An SE^2 must have interactive system capabilities. [Ref. 85, 84, 94, 95, 98]. There are two reasons for this: interactive systems aid communication among the participants in design, and designers can work at their own pace (interactively) rather than someone else's (batch). User participation, one of the development tasks, is simplified when using interactive systems.

8. Recent Developments

Two ideas about SE^2s, personal development systems and a software engineering knowledge base, seem to unify the three development tasks and embody the characteristics just

stated. Personal development systems have all of the characteristics discussed (integrated, uniform, support a solution strategy, adaptable, functionally unique, and interactive). Their most important feature, though, is the dedicated support to a single designer [Ref. 89 , 94 , 95]. A software environment knowledge base would capture information about the design activity (for example, design decisions) as well as the development process (a continuous effort) for managers, designers, and users [Ref. 96]. This knowledge base would make the information easily available and would be done automatically.

C. A SOFTWARE ENGINEERING ENVIRONMENT FOR PROTOTYPES

Most authors agree that a 'successful' SE^2 must support a certain view of the design process [Ref. 85, 94, 95, 97]. Following the lead of Lauber [Ref. 85], a collection of tools, or components, which support the set of seven design elements of Chapters II and III, and which support the development of prototypes, covered in Chapter IV, is presented. This is followed by descriptions of how such components support software design principles and prototyping.

1. Technical Components

There are several components which should be included in an SE^2 [Ref. 62, 75, 79, 83, 101].

a. Database Management Systems (DBMS)

A DBMS serves two purposes in an SE^2. First, the DBMS enables storing and retrieving information about the design as well as the development process. For example, a record could be kept of when each version of the prototype was released, who designed it, relevant design decisions,

and so on. Second, a DBMS allows for 'quick' design and programming of data handling features [Ref. 62, 61, 83]. Recall that the ability for quick turnaround of a working system to the user is a necessary feature in many prototyping situations.

b. Generalized Input and Output Software

Query languages, report generators, and report writers are often features of a DBMS (for example, FOCUS, RAMIS II, and NOMAD provide these features). These features allow for easy data retrieval and data update. Report generators can produce complicated reports with minimal programming effort [Ref. 61, 62, 79, 101].

c. Graphics Tools

Graphics are ideal for representing the large, and often complex, structures of non-trivial software designs. These tools are particularly suited for the methodologies which use structure charts. For example, Delisle [Ref. 102] describes a set of graphics-based tools, an Edit tool, an Evaluate tool, a Format tool, and a Clean-up tool, which were developed to support Structured Analysis).

d. High-level Languages

High-level languages (variously described as non-procedural languages, formal specification languages, and so on) have one objective, flexibility [Ref. 62, 83, 101]. Such languages enable the designer to describe" what to do" rather than "how to do" it. The system resolves the procedure and should produce executable machine code. The designer, given such a tool, can use abstraction to its fullest extent (the Gamma software engineering system [Ref. 103], for example, specifically supports abstraction).

e. Interactive Systems

Devices and equipment (for example, working stations) which support interaction are essential [Ref. 61, 62, 83, 98]. Interactive terminals give users and designers the perception of rapid and efficient operation and revision. Generally, these facilities are adapted from the host computer or network of the SE². (Personal development systems could be thought of as extensions of interactive systems.)

f. Application-oriented Models

Models are an important feature of an SE². They are used to support human decision making [Ref. 61, 62]. Examples of models which are potentially useful are financial models (as in FOCUS) or simulation models. Real-world modelling [Ref. 43] is also an important element in the SE².

g. Tools for Software Testing

There is clearly a need for tools which simplify software testing [Ref. 83, 101]. Hausen and Muellerburg report that most tools of this type concentrate on verification and validation, that is convincing ourselves that the program will execute properly. They argue that software tools for program testing should cover more than just verification and validation. They recommend a philosophy of quality improvement which includes quality assurance (defining software standards and controlling their observation), acceptance testing (demonstrating to the user that the software is acceptable for operation), and verification and validation.

2. Support for Software Design

Any SE^2 must be based on a particular view of software design. [Ref. 85, 94, 95, 97]. The view presented in Chapters II and III is unique, although elements of that view may be supported in different ways by different systems.

The SE^2 must recognize, and provide facilities for, the symmetrical and adaptable process of design. If the solution to a problem changes the problem, features of the SE^2 must allow revision, interactive use by clients (it is their problem, after all), and record-keeping, especially of decisions.[9]

The satisficing aspect of design may best be met by using the modelling tools of the SE^2. Simulation tools can help answer "what if" and performance questions. Financial models can help decide economic questions. Planning, control, and estimating models can also help to decide on the worth of various tradeoffs.

The "wicked problem" aspect is particularly vexing in the SE^2. High-level languages can help by allowing an abstract description as a formulation of the problem. The abstract statements are then transformed by the system into concrete (that is, executable) code [Ref. 105, 106, 107].

Communications between the user and the designer is aided by interactive systems. Graphics also aid user (and designer) comprehension. Alexander and others have shown how the notion of patterns helps bridge the communication gap. Kuc, and others, [Ref. 80, 84, 108, 109, 110, 111] have adopted this concept in their "forms-based" software development environment. The 'forms' within the system are

----

[9] White [Ref. 104] presents a model for recording relevant information about design decisions during software development.

used to identify and define 'patterns' that are above the level of programming language constructs. Although a full discussion of the TRIAD (TRee-based Information Analyzer and Developer) system is beyond the scope of this work, it is an excellent candidate for an SE² which supports software prototyping.

The interactive facilities and modelling features of the SE² will help to aid the learning process in design. The notion of 'learning by doing' was introduced in Chapter III. To support that notion, the SE² should allow the designer to learn, early, the consequences of a design decision. The designer must then be given the chance to revise his decision, based on the 'operation' experience.

Organizational issues must be explicitly recognized in any SE². First, there are organizational resources which are needed to support the SE²: programmers, operators, managers, space and facilities, and the computer hardware assocated with the SE². Second, the work patterns and work skills of the people who work in the SE² are likely to change. Unfortunately, most current development environments stress the environment over the users of the environment [Ref. 98]. Typically, those environments have "quirks" which require people to adjust. The system should adjust to the skills and the preferences of the designers who use it (using, for example, custom default features). If we consider the SE² as an element of a complex organization [Ref. 59, 60, 98], the environment's interaction with people is crucial; without that interaction, the SE² is useless in any practical sense.

Finally, the SE² must explicitly recognize the evolutionary aspect of software design. The current systems support the waterfall model of software development [Ref. 94, 95]. The database management system, interactive facilities, and high-level languages will easily support the

evolutionary concept of design. Report generators and report writers should aid the documentation process as the design evolves.

3. Support for the Prototype Process

The process of developing a software prototype was covered in Chapter IV. There are four steps in that process: 1) identifying the user's basic requirements, 2) developing a working prototype, 3) implementing and using the protctype, and 4) revising and enhancing the prototype.

An existing database of the SE^2 is ideal for identifiying the user's initial requirements. However, there are problems if the database is empty. Kangasallo [Ref. 112] presents a model in which information requirements are interpreted as a set of complex queries by the database management system. Additional features of that model include a 'program constructor' which generates code based on the queries. A working prototype is a result of this model.

Another method depends not only on the database management system but also on the automated tools within the SE^2. Cheatham [Ref. 105] presents a system in which the designer and user develop an abstract model of the problem (possibly from the database). Transformation refinement is applied (by the automated tools) which results in executable code--a working prototype.

In both of these instances, the SE^2 supports the development of the user's basic requirments followed by an automated process of developing a working prototype. It is important that some effort be made to analyze the user's requirements so that reasonable queries can be made and reasonable models (of the problem) can be developed.

Other systems are available which help to develop a basic set of user requirements. Some are quite complex [Ref. 32] and might be difficult to integrate with the SE².

Developing a working prototype, quickly, should not be difficult to accomplish in the SE². High-level languages; code generators; transformation refinement (mentioned above); application development systems, such as ACT/1 [Ref. 76] and, application generators [Ref. 75] make it easier to develop working prototypes. Ideally, the system would be completely automated.

An abstract model allows the designer to focus more easily on the results of his or her decisions, rather than the implementation details. An abstract model also promotes flexibility when it is reused. [Ref. 105]

Implementing and using the prototype becomes much easier when interactive systems are used. User interaction is essential and interactive terminals allow the user to perceive rapid operation and revision. They also help to speed user evaluation [Ref. 62].

Revision and enhancement are facilitated in the SE² by using the database management system, high-level languages (and abstract models), the generalized input and output tools, and graphics tools. The database contains a record of past designs and design decisions, changes are easily made to abstract models and high-level language constructs, default values of the generalized input and output tools are easily adjusted, and the graphics tools will enable both users and designers to spot patterns quickly. The user is quickly accommodated, the database management system automatically tracks versions and design decisions, and the designer is able to defer low-priority details without fear of compromising the design: the SE² relieves the designer of much, if not all, of the drudgery normally associated with software design.

D. SUMMARY

The preceding sections have reviewed the characteristics needed in a software engineering environment, have identified the components of a software engineering environment, and have described how the components interrelate to support both software design and the prototype process.

It is doubtful that there are any software engineering environments which support completely the idea of prototyping. To a limited degree, commercial systems, such as FOCUS, NOMAD, ACT/1, to name a few, support particular aspects of the prototype process. For example, FOCUS and NOMAD facilitate applications programming in the business community by allowing the designer to customize reports or other appplications for a specific user, or group, based on an already existing database--the vice-president of sales might be interested in the sales of a particular product in a particular geographical area. ACT/1, and other similar products, make it easier for designers to customize the formats of terminal screens for the user.

The products mentioned here are three of several hundred commercial and research systems and environments. This chapter has purposely avoided a lengthy review of any of those hundreds, and mentions a few by way of example only.

VI. CASE EXAMPLES

The four cases which follow were chosen because in each there was an explicit decision began to develop and use software prototypes before the project began.

A. SYMMETRY, EVOLUTION, SATISFICING, AND COMMUNICATION

Heckel [Ref. 113] describes the process of developing a prototype while designing the Craig translator. The project team explicitly chose to develop prototypes for several reasons. First, they were concerned about the problems which users would <u>actually</u> experience, rather than those problems which the designers <u>imagined</u> might be important. This concern is directly related to the symmetry aspect in design. That is, the solution and problem interrelate such that the solution depends critically upon the context of the problem. In this case, the context is the consumer's use of the Translator. If the product does not perform as "expected", it will not sell.

Second, the project team was interested in postponing decisions about restraints on the final system until they had to. In other words, their design <u>evolved</u>. The designers ignored certain restrictions which had been placed on memory size, as long as they carefully considered the effects of their decisions on the production version of the Translator.

Third, the project team planned to use the prototype as the software specification. Because they had two "versions" of the prototype, a black box translator and the program listing, they thought that they would avoid the traditional misunderstandings and contradictions often found in written

software specifications. In this case, the designers were concerned about communications, not only between the "user" and the "designer" but also among themselves.

Heckel's description shows that the prototypes (there were 30 versions!) were used to clarify requirements and to verify the feasibility of the design. Heckel states that if they had been forced to make a particular design decision earlier than they did, they probably would have made a less satisfactory decision.

The project was judged a success, although progress seemed slow and painful. Heckel identifies four benefits of developing prototypes:

1. The project team could keep trying new things;
2. The prototype was a good model of the final product, so everyone had similar expectations about what the product would do;
3. Several decisions could be postponed without affecting the schedule; and,
4. The designers focused their efforts on opportunities rather than problems.

The development process had some disappointments: software development took longer than expected and the final product took more memory than expected. Heckel did not speculate on whether these "disappointments" could have been avoided. One interpretation is that the designers were unable to meet all of their objectives and when time ran out their design was judged to be good enough. Thus, the "disappointments" can be attributed to the satisficing aspect of design, especially the need for more memory. The designers obviously made a trade-off between the "goodness" of the product and the amount of memory they had originally planned.

This case illustrated how the use of prototypes addresses the symmetry, evolution, communications, and satisficing aspects of design.

B. LEARNING

Hemenway and McCusker [Ref. 116] describe an exploratory project which is leading to the development of an order negotiation and entry support system for telephone service (the Bell system). The project is the development of the user interface and the supporting software for the system.

There are two reasons given for building an operational prototype: 1) to evaluate the user interface and 2) to assess the feasibility of a particular software architecture. Even though the reasons coincide with two uses of prototypes (that is, to clarify user requirements and to verify the feasibility of a design) they are related to two aspects of design. The aspects are learning and communication between the designer and user.

Prototypes of the software were developed to determine whether a table-driven system could be designed. Prototypes of the user-interface were used to determine whether the user-interface would substantially increase the length of time service representatives spend on orders (compared to manual order entry and search).

The case concludes by stating that the results of the prototype evaluation led to making several recommendations to the designers of the first release of the system. *Hence, the prototype served to help the designers learn more about their solution and their problem.*

C. WICKED PROBLEMS, COMMUNICATIONS, AND THE ORGANIZATIONAL CONTEXT

Jenkins [Ref. 114] discusses how the decision to develop a prototype led to successful development of an automated data processing facility for the Congressional Budget Office.

Two aspects of software design are apparent in this case: 1) communications between users and designers and 2) the organizational context of the system. Communications between the designers and users was greatly improved by using a prototype. Rather than try to decide on the designer's effectiveness by reviewing written specifications, managers witnessed operating demonstrations. The prototype also showed non-technical users what it was possible to do in their application areas with the new tools.

By far the most important aspect illustrated by this case, is the concern of the designers for organizational issues. The Congressional Budget Office serves the needs of the Congress, admittedly a complex organization. So, the designers needed immediate responses to Congressional inquiries, because when information is needed, it is often needed immediately or its value is lost.

This organizational aspect is also closely related to wicked problems. Recall that wicked problems refer to social system problems which are ill-formulated, where the information is confusing, where there are many clients and decision-makers with conflicting values, and where the

ramifications in the whole system are thoroughly confusing. Clearly Congress is faced with these kinds of problems. There is every reason to expect that the Congressional Budget Office deals with similar problems when responding to Congressional inquiries.[10]

<u>The case presented by Jenkins illustrates how prototypes can aid software design when faced with critical organizational issues and wicked problems</u>.

D. COMMUNICATION, LEARNING, AND EVOLUTION

Groner and others [Ref. 115] present a case of using prototypes to clarify the user's requirements. The case is unusual because it started with a proposal from outside the user's community. The designers set out to determine if and how computer technology could meet the information processing needs of medical researchers.

This case is a clear illustration of the importance of communications between the designer and the user and the representation used for communicating.

> Prototypes were required in the requirements analysis phase because without concrete, working examples our potential users could not be sure that computer systems are needed, what functions they should perform, or how they would use them. [Ref. 115 : p. 100]

Less clearly stated is the implication of learning during the design process. The intitial design of the prototype was based on the designer's knowledge about

---

[10]Consider the fluctuations from Congress to Congress, chairman to chairman, committee to committee; from year to year, week to week, and even from hour to hour during the Budget Committee markup sessions [Ref. 114 : p. 22].

information processing needs for medical research. Subsequent versions were improved based on use by and comments from clinical researchers. The project participants

> . . . agreed to learn about each other's disciplines, then define problems and attempt to devise and evaluate solutions in collaboration with others in the target user community. [Ref. 115 : p. 101]

The project used an incremental implementation strategy (evolution) under which major software releases were scheduled approximately every four months. Several hundred software changes were made over a period of a year and half. This case shows how prototypes can be used to create the final system.[11]

The case presented by Groner and others is an excellent example of how communications, learning, and evolution are intertwined in software design. The development of prototypes helped all of the design participants cope with those aspects of software design.

E. SUMMARY

These cases illustrate how prototypes help designers cope with the seven aspects of design which were covered in Chapters II and III. In each of the cases, the authors point to success. For Heckel, the prototypes led to a product that was easy to use, had a number of useful features, and was implemented on a single-chip microprocessor.

---

[11]The case description leads the reader to think that a "production" system was not developed. Every indication is that the prototypes evolved into the production system.

Hemenway and McCusker say only that prototype evaluation led to recommendations to the designers. From this, we can safely infer that the prototype aided the designer's under-standing of the problem.

For Jenkins, the overall assessment to the prototype was positive. Managers liked the idea of a prototype because there was no prior commitment to a particular course of action.

Groner and others believe that the greatest benefit of the prototype is that the prototypes are concrete, working examples of computer systems which are meeting everyday needs.

VII. CONCLUSIONS

A new view of design was presented in Chapter II. This view identifies a set of seven interrelated and mutually dependent elements which were found in the literature. Support for these elements was found throughout the computer and information science literature. The set of seven elements explains how best to cope with the problems, ambiguity, and uncertainty associated with software design.

The process of developing a software prototype is presented as the most appropriate way to incorporate the design elements into software design. In fact, the prototype process exploits certain elements, such as communication between the user and designer, to improve the overall design of the software.

One of the more important conclusions is that software designers, especially designers of large-scale systems, have much to learn from designers in other fields. The software design literature shows little evidence of influence from other design fields. This work is a start toward that needed transfer of knowledge.

The software prototype may be the sensible way to design large-scale systems. Recall that complex design problems have been called wicked problems. If some large-scale system developments are 'more wicked' than others, then developing prototypes seems to be the only way to design the system.

Software prototyping enables users and designers to cope with ill-defined problems and changing requirements. Past experience indicates that bad technical engineering is not a problem with software development. Rather, unsatisfactory design decisions and faulty information are the real

problems. Software prototypes provide a mechanism which allows designers to test their decisions and to learn more about the problem. The prototypes also allow users a construcyive environment in which to express their satisfaction or dissatisfaction and a stimulant in learning how to deal with their problems.

Software prototypes, however, present special difficulties because they are not the universal remedy for software design problems. Careful management is needed to ensure the software ptototype is really designed and not just put together. Careful thought and planning are necessary before coding begins. Managers, designers, and users must remember that a software prototype is an experiment. Judgement and commitment are needed to control endless iterations. Managers must have the wisdom to know when to stop. Often, while developing successive prototypes, there is a tendency to delay formally documenting the system. While this problem is not unique to prototypes, there must be attentive management and commitment to ensure adequate and complete documentation.

In spite of these cautions, evidence indicates that developing and using software prototypes is the best option for coping with software design problems, for ensuring the system is delivered, and for ensuring a satisfied user population.

VIII. RECOMMENDATIONS FOR FURTHER STUDY

A. MANAGEMENT

Developing software prototypes presents management with some unusual problems. Many of our current management techniques depend on getting the project done right the first time [Ref. 117]. As we are well aware, this seldom occurs. Research is needed to assess the effect of prototype development on management.

1. How does the manager decide when to cease development of prototypes? When is the project ended?
2. How do managers deal with increased communications between users and designers? If special management controls are needed, how far should they go?
3. What management style best suits managers of software prototype projects?
4. How is the project budgeted and controlled? How is progress measured?

B. ACQUISITION AND CONTRACT MANAGEMENT

Current acquisition and contract management procedures and regulations for software appear to be less than satisfactory, within the Federal Government generally, and the Department of Defense particularly. Even as these procedures and regulations are changing, there is some evidence that the traditional model of software development may become required. The Department of Defense has begun to address the concept of software prototypes in the DoD Software Technology Initiatives [Ref. DoD81 : p. 69-71], but this research appears to be concerned only with requirements specifications.

1. How can or how should acquitition and contract management procedures and regulations accomodate the principles of design and software prototypes?
2. What is the best strategy for encouraging acceptance of the software design principles and software prototypes?
3. How might the elements of software design and developing software prototypes help with the acquisition and contract management for embedded computer resources?

C. ORGANIZATIONAL CONTEXT

Kling and Scacchi [Ref. 59, 60] reviewed a large number of organizational studies while developing their views about the effect of computer systems upon organizations. When their ideas are considered within the context of software prototypes, further research is needed.

1. How will changes in theories of organizational development affect the process of developing prototypes?
2. Is any one organizational theory best suited for software design and software prototypes?
3. What are the social dynamics of software design?
4. What are the social dynamics of developing software prototypes?

D. QUALITY

A fundamental part of design is to satisfy the needs for quality. Rooke [Ref. Rook82] has concluded that design is the most important factor in determining overall quality. Even though one of the objectives of developing software prototypes is to achieve user satisfaction (a major element of quality), research is needed to determine how prototypes can affect software quality.

1. If we accept that prototypes will affect a change in software technology, how will that change influence our perceptions of quality? That is, will software prototypes lead users to expect more than can be met?
2. How might the concept of Quality Circles fit the process of developing software prototypes?
3. To what extent will software prototypes influence software quality? Since prototyping requires concensus, who is ultimately responsible for product quality and liability? Should anyone be "ultimately" responsible?

E. REPRESENTATION

The software prototype is the ultimate representation of the user's requirements. The written specification anchors the other end of the representations scale.

1. What other types of representations can aid software design and the development of software prototypes?
2. What methods are suitable for representing abstractions when identifying a user's requirements before developing a software prototype?
3. How do different representations affect our perceptions and real world knowledge? Can different, initial representations lead to quicker design and development of software prototypes?

LIST OF REFERENCES

1. Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

2. Thayer, Thomas A., Lipow, Myron, and Nelson, Eldred C., *Software Reliability*, TRW Series of Software Technology, Vol. 2, North-Holland Publishing Co., Amsterdam, 1978.

3. Boehm, Barry W. and others, *Characteristics of Software Quality*, TRW Series of Software Technology, Vol. 1, North-Holland Publishing Co., Amsterdam, 1978.

4. Peters, Lawrence J., *Software Design: Methods and Techniques*, Yourdon Press, New York, 1981.

5. Dunn, Robert and Ullman, Richard, *Quality Assurance for Computer Software*, McGraw-Hill, New York, 1982.

6. Alexander, Christopher, *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, MA, 1964.

7. Archer, L. Bruce, "An Overview of the Structure of the Design Process," *Emerging Methods in Environmental Design and Planning*, Gary T. Moore, ed., MIT Press, Cambridge, MA, p. 285-307, 1970.

8. Jones, J. Christopher, *Design Methods, Seeds of Human Futures*, Wiley Interscience, John Wiley & Sons, Ltd., London, 1970.

9. Churchman, C. West, "Wicked Problems," *Management Science*, vol. 14, no. 4, December 1967, p. B-141-B142.

10. Alexander, Christopher and others, *Pattern Language*, Oxford Press, 1974.

11. Page, J. K., "A Review of the Papers Presented at the Conference," *Conference on Systematic and Intuitive Methods in Engineering, Industrial Design, Architecture, and Communications*, J. C. Jones and D. G. Thornley, eds., The Macmillan Company, New York, 1963, p. 205-215.

12. Ellinger, John Henry, *Design Synthesis, Vol. 1*, John Wiley & Sons, Ltd. London, 1968.

13. Rittel, Horst, "Some Principles for the Design of an Educational System for Design," Journal of Architectural Education, v 26, nos 1-2, Winter-Spring, 1971/1972, p. 16-26.

14. Simon, Herbert, The Sciences of the Artificial, MIT Press, Cambridge, MA, 1981.

15. Bazjanac, Vladimir, "Architectural Design Theory: Models of the Design Process," Basic Questions of Design Theory, William B. Spiller, ed., American Elsevier Publishing Co., NY, p. 3-19, 1974.

16. Cross, Nigel, Naughton, John, and Walker, David, "Design Method and Scientific Method," Design Studies, v. 2, n. 4, Oct. 1981, p. 195-201.

17. Smithies, K. W., Principles of Design in Architecture, Van Nostrand Reinhold Co., New York, 1981.

18. Popper, Karl Raimund, Objective Knowledge, An Evolutionary Approach, Oxford University Press, London, 1972.

19. Polya, G., How To Solve It, A New Aspect of Mathematical Method, 2nd edition, Princeton University Press, Princeton, New Jersey, 1957.

20. Dodd, W. P., "Prototype Programs," Computer, v 13, n 2, February 1980, p. 81.

21. U. S. Department of Defense, Candidate R & D Thrusts for the Software Technology Initiative, foreword by Joseph C. Batz, Office of the Under Secretary of Defense for Research and Engineering (Electronics and Physical Sciences), May 1981.

22. Madnick, Sturat E. and Donovan, John J. Operating Systems, McGraw-Hill, New York, 1974.

23. Peters, Lawrence, "Relating Software Requirments and Design", Software Engineering Notes vol. 3 no. 5, November 1978, p. 67 -71.

24. Podolsky, Joseph L., "Horace Builds a Cycle", Datamation, vol. 23, no. 11, November 1977, p. 162-168.

25. Voight, Susan, "Program Design by a Multidisciplinary Team", Proceedings of the First International Conference on Software Engineering, IEEE Computer Society, 1975, p. 63-69.

26. Conn, Alex Paul, "Maintenance: A Key Element in Computer Requirements Definition", Proceedings of the Computer and Software Applications Conference, 1980, p. 401-406.

27. McCracken, D. D., and Jackson, Michael A., "Life-Cycle Concept Considered Harmful", Software Engineering Notes, vol. 7, no. 2, April 1982, p. 29-32.

28. Neumann, Peter G., "Software Evolution and the Dimensions of Change", Letter from the Editor, Software Engineering Notes, vol. 6, no. 1, January 1981, p. 1.

29. Land, Frank, "Adapting to Changing User Requirements", Information and Management, vol. 5, 1983, p. 59-75.

30. Rauch-Hinden, Wendy, "Some Answers to the Software Problems of the 1980s", Data Communications, vol. 10, no. 5, May 1981, p. 57-70.

31. Lockett, JoAnn, "Using Performance Metrics in System Design", Software Engineering Notes, vol. 3, no. 5, November 1978, p. 156-159.

32. Zave, Pamela, "An Operational Approach to Requirements Specifications for Embedded Systems", IEEE Transactions on Software Engineering, vol. SE-8, no. 3, May 1982, p. 250-269.

33. Canavan, Edward M., "Systems, Relity and the Systems Practioner", Journal of Systems Management, January 1981, p. 26-28.

34. Gilb, Tom, "High-Level Systems Architecture: Design by Objectives", Computer, vol. 13, no. 5, May 1980.

35. Wasserman, Anthony Ira, "A Top-Down View of Software Engineering", Proceedings of the First International Conference on Software Engineering, IEEE Computer Society, 1975, p. 1-7.

36. Brittan, J. N. G., "Design for a Changing Environment", The Computer Journal, vol. 23, no. 1, February 1980, p. 13-19.

37. Peters, Lawrence J. and Tripp, Leonard L., "Is Software Design 'Wicked'?", Datamation, vol. 22, no. 5, May 1976, p. 127+.

38. Scharer, Laura L., "Pinpointing Requirements," Datamation, vol. 27, no. 4, April 1981, p. 139-151.

39. Horning, J. J., "Program Specification: Issues and Observations," Program Specification, J. Staunstrup, ed., Lecture Notes in Computer Science, vol. 134, Springer-Verlag, Berlin, 1982, p. 5-18.

40. Chafin, Roy I., "The System Analyst and Software Requirements Specifications", Proceedings of the Computer and Software Applications Conference, 1980, p. 254-258.

41. King, William R., and Rodriquez, Jamie I., "Participative Design of Strategic Decision Support Systems: An Empirical Assessment", Management Science, vol. 27, no. 6, 1981, p. 717-726.

42. Robey, Daniel and Farrow, Dana, "User Involvement in Information System Development: A Conflict Model and Empirical Test", Management Science, vol. 28, no. 1, January 1982, p. 73-85.

43. Greenspan, Sol J., Mylopoulos, John, and Borgida, Alex, "Capturing More World Knowledge in the Requirements Specification", Proceedings of the Sixth International Conference on Software Engineering, IEEE Computer Society Press, Silver Spring, Maryland, 1982, p. 225-234.

44. Gilb, Tom, "Evolutionary Development", Software Engineering Notes, vol. 6, no. 2, April 1981, p. 17.

45. Stavely, Allan M., "Design Feedback and its Use in Software Design Aid Systems", Software Engineering Notes, vol. 3, no. 5, November 1978, p. 72-78.

46. Brooks, Frederick P., Jr., The Mythical Man-Month, Essays on Software Engineering, Addison-Wesley Co., Reading, Massachusetts, 1975.

47. Lehman, Meir M., "Laws and Conservation in Large-Program Evolution," Second Software Life Cycle Management Workshop, 20-22 August 1978, p. 140-145.

48. Frank, James W., "Applications Design by Trial and Error", Infosystems, September 1979, p. 76-78.

49. Hall, Patrick A. V., "In Defense of Life Cycles", Software Engineering Notes, vol. 7, no. 3, July 1982, p. 23.

50. Lawrence, M. J., "An Examination of Evolution Dynamics", Proceedings of the Sixth International Conference on Software Engineering, IEEE Computer Society Press, Silver Spring, Maryland, 1982, p. 188-196.

51. Urban,G. L. and Karash, R., "Evlutionary Model Building", Journal of Marketing Research, vol. 8, 1971.

52. Swartout, William and Balzer, Robert, "On the Inevitable Intertwining of Specification and Implementation," Communications of the ACM, vol. 25, no. 7, July 1982, p. 438-440.

53. Zmud, R. W. and Cox, J. F., "The Implementation Process: A Change Approach", MIS Quarterly, vol. 3, June 1979, p. 35-43.

54. Boland, Richard J. Jr., "The Process and Product of System Design", Management Science, vol. 24, no. 9, May 1978, p. 887-898.

55. Alavi, Mayram and Henderson, John C., "An Evolutionary Strategy for Implementing a Decision Support System", Management Science, vol. 27, no. 11, November 1981, p. 1309-1323.

56. Blum, Bruce I., "The Life Cycle--A Debate over Alternative Models", Software Engineering Notes, vol. 7, no. 4, October 1982, p. 18-20.

57. Zvegintzov, Nicholas, "What Life, What Cycle?", AFIPS Conference Proceedings, National Computer Conference, Volume 51, 1982, p. 561-568.

58. Gladden, G. R., "Stop the Life-Cycle, I Want to Get Off", Software Engineering Notes, vol. 7, no. 2, April 1982, p. 35-39.

59. Kling, Rob and Scacchi, Walt, "Computing as Social Action: The Social Dynamics of Computing in Complex Organizations," in Advances in Computers, Volume 19, Marshall C. Yovits, ed., Academic Press, NY, 1980, p. 249-327.

60. Kling, Rob and Scacchi, Walt, "The Web of Computing: Computer Technology as Social Organization," in Advances in Computers, Volume 21, Marshall C. Yovits, ed., Academic Press, NY, 1982, p. 1-90.

61. Berrisford, Thomas and Wetherbe, James, "Heuristic Development: A Redesign of Systems Design", MIS Quarterly, vol. 3, no. 1, March 1979, p. 11-19

62. Naumann, Justus D. and Jenkins, A. Milton, "Prototyping: The New Paradigm for Systems Development", MIS Quarterly, September 1982, p. 29-44.

63. Rich, Charles and Waters, Richard C., "The Disciplined Use of Simplifying Assumptions," Software Engineering Notes, vol. 7, no. 5, December 1982, p. 150-154.

64. Earl, Michael J., "Prototype Systems for Accounting, Information and Control", Accounting, Organizations and Society, vcl. 3, no. 2, 1978, p. 161-170.

65. Basili, Victor R. and Turner, Albert J., "Iterative Enhancement: A Practical Technique for Software Development", First International Conference on Software Engineering, IEEE Computer Society, 1975, p. 56-62.

66. Asner, Michael, King, Alan and Darke, Raymond G., "Prototyping: A Low Risk Approach to Developing Complex Systems, (Part 2--Methodology)", Business Quarterly, vol. 46, no. 4, Winter 1981, p. 34-38.

67. McCracken, Daniel D., "A Maverick Approach to Systems Analysis and Design," Systems Analysis and Design: A Foundation for the 1980's, William W. Cotterman and others, eds., Elsevier Science Publishing Co., New York, 1982, p. 446-451.

68. Bally, Laurent, Brittan, John, and Wagner, Karl H., "A Prototype Approach to Information System Design and Development", Information and Management, vol. 1, 1977, p. 21-26.

69. Weiser, Mark, "Scale Models and Rapid Prototyping", Software Engineering Notes, vol. 7, no. 5, December 1982, p. 181-185.

70. Barstow, David, "Rapid Prototyping, Automatic Programming, and Experimental Sciences:, Software Engineering Notes, vol. 7, no. 5, December 1982, p. 33-34.

71. Blum, Bruce I., "Rapid Prototyping of Information Management Systems", Software Engineering Notes, vol. 7, no. 5, December 1982, p. 35-38.

72. Davis, Alan M., "Rapid Prototyping using Executable Requirements Specifications", Software Engineering Notes, vol. 7, no. 5, December 1982, p. 39-44.

73. Feather, Martin S., "Mapping for Rapid Prototyping", Software Engineering Notes, vol. 7, no. 5, December 1982, p. 17-24.

74. Cohen, Donald, Swartout, William and Balzer, Robert, "Using Symbolic Execution to Characterize Behavior", Software Engineering Notes, vol. 7, no. 5, December 1982, p. 25-32.

75. Canning, Richard W., ed., "Developing Systems by Prototyping", EDP Analyzer, vol. 19, no. 9, September 1981.

76. Mason, R. E. A. and Cary, T. T., "An Approach to Prototyping Interactive Information Systems", Communications of the ACM, vol. 26, no. 5, May 1983, p. 346-354.

77. McCoyd, Gerard C. and Mitchell, John R., "System Sketching: The Generation of Rapid Prototypes for Transaction Based Systems", Software Engineering Notes, vol. 7, no. 5, December 1982, p. 127-132.

78. Taylor, Tamara and Standish, Thomas A., "Initial Thoughts on Rapid Protoyping Techniques", Software Engineering Notes, vol. 7, no. 5, December 1982, p. 160-166.

79. Asner, Michael and King, Alan R., "Prototyping: A Low-Risk Approach to Devleoping Complex Systems", Business Quarterly, vol. 46, no. 3, Autumn 1981, p. 30-34.

80. Ramanathan, J. and Shubra, C. J., "Use of Anotated Schemes for Developing Prototype Programs," Software Engineering Notes, vol. 7, no. 5, December 1982, p. 141-149.

81. Heitmeyer, C., Landwehr, C. and Cornwell, M., "The Use of Quick Prototypes in the Secure Military Message Systems Project", Software Engineering Notes, vol. 7, no. 5, December 1982, p. 85-87.

82. Spiegel, Mitchell G., "Prototyping: An Approach to Information and Communication System Design", Performance Evaluation Review, vol. 10, no. 1, Spring 1981, p. 9-19.

83. Hausen, Hans-Ludwig and Muellerburg, Monika, "Architecture of Software Systems in theContext of Software Engineering Environments," Systems Architecture, Proceedings of the Sixth ACM European Regional Conference, IPC Science and Technology Press Limited, Surrey, England, 1981, p. 147-157.

84. Soni, Dilip, "Design and Modeling of TRIAD, an Adaptable, Integrated Software Environment," Computer Science Guest Lecture, Naval Postgraduate School, Monterey, CA, March 1983.

85. Lauber, Rudolf, "Development Support Systems," Computer, vol. 15, no. 5, May 1982, p. 36-46.

86. Wasserman, Anthony I., "Automated Development Environments," Computer, vol. 14, no. 4, April 1981, p. 7-10.

87. Kernighan, B. W. and Plauger, P. J., "Software Tools," First International Conference on Software Engineering, IEEE Computer Society, 1975, p. 8-13.

88. Kernighan, Brian W. and Mashey, John R., "The Unix Programming Environment," Computer, vol. 14, no. 4, April 1981, p. 12-24.

89. Gutz, Steve, Wasserman, Anthony I., and Spier, Michael J., "Personal Development Systems for the Professional Programmer," Computer, vol. 14, no. 4, April 1981, p. 45-53.

90. Barstow, David R. and Shrobe, Howard E., "Guest Editorial: Programming Environments," IEEE Transactions on Software Engineering, vol. SE-7, no. 5, September 1981, p. 449-450.

91. Teitelman, Warren and Masinter, Larry, "The Interlisp Programming Environment," Computer, vol. 14, no. 4, April 1981, p. 25-33.

92. Wegner, Peter, "The Ada Language and Environment," Software Engineering Notes, vol. 5, no. 2, April 1980, p. 8-14.

93. U. S. Department of Defense, "STONEMAN," Requirements for Ada Programming Support Environments, February 1980.

94. Wasserman, Anthony I., "Toward Integrated Software Development Environments,: Scientia, vol. 115, 1980, p. 663-684.

95. Wasserman, Anthony I., "Automated Tools in the Information System Development Environment," Automated Tools for Information Systems Design, H.-J. Schneider and A. I. Wasserman, eds., North-Holland Publishing Co., Amsterdam, 1982, p. 1-9.

96. Rajaraman, M. K., "A Characterization of Software Design Tools," Software Engineering Notes, vol. 7, no. 4, October 1982, p. 14-17.

97. U. S. Department of Defense, Ada Joint Program Office, Ada Methodologies: Concepts and Requirements, November 1982.

98. Prentice, Dan, "An Analysis of Software Development Environments," Software Engineering Notes, vol. 6, no. 5, October 1981, p. 19-27.

99. Korzybski, Alfred, Science and Sanity, An Introduction to Non-Aristotelian Systems and General Semantics, 4th edition, preface by Russell Meyers, M.D., The International Non-Aristotelian Library Publishing Co., Lakeville, Connecticut, 1958.

100. Barr, Avron and Feigenbaum, Edward A., ed. The Handbook of Artificial Intelligence, Vol. II, William Kaufmann, Inc., Los Altos, California, 1982.

101. Ewers, Jack and Vessey, Iris, "The Systems Development Dilemma--A Programming Prespective," _MIS Quarterly_, June 1981, p. 33-45.

102. Delisle, Norman M., Menicosy, David E., and Kerth, Norman L., "Tools for Supporting Structured Analysis, _Automated Tools for Information Systems Design_, H.-J. Schneider and A. I. Wasserman, eds., North-Holland Publishing Co., Amsterdam, 1982, p. 11-20.

103. Falla, M. E., "The Gamma software engineering system," _The Computer Journal_, vol. 24, no. 3, 1981, p. 235-242.

104. White, John R., "A Decision Tool for Assisting with the Comprehension of Large Software Systems," _Automated Tools for Information Systems Design_, H.-J. Schneider and A. I. Wasserman, eds., North-Holland Publishing Co., Amsterdam, 1982, p. 49-65.

105. Cheatham, Thomas E., Jr., "Programming Support Environments," Computer Science Guest Lecture, Naval Postgraduate School, Monterey, CA, December 15, 1982.

106. Lundberg, Bengt, "IMT--An Information Modelling Tool," _Automated Tools for Information Systems Design_, H.-J. Schneider and A. I. Wasserman, eds. North-Holland Publishing Co., Amsterdam, 1982, p. 21-30.

107. Cheatham, Thomas E., Jr., "Comparing Programming Support Environments," _Software Engineering Environments_, North-Holland Publishing Co., Amsterdam, 1981, p. 11-25.

108. Kuo, Jeremy and others, "An Adaptable Software Environment to Support Methodologies," Technical Report TRIAD-2, Department of Computer and Information Science, Ohio State University, January 1983.

109. Kuo, H. C., Li, C. H., and Ramanathan, J., "A Form-Based Approach to Human Engineering Methodologies," _Proceedings of the 6th International Conference on Software Engineering_, IEEE Computer Science Press, 1982, p. 254-263.

110. Kuo, H. C., and others, "System Architecture of an Adaptable Software Environment," Department of Computer and Information Science, Ohio State University, Technical Report, TRIAD-TR-1-83.

111. Ramanathan, J. and Soni, D., "Design and Implementation of an Adpatable Software Environment," to be published in the _Journal of Computer Languages_.

112. Kangasallo, Hannu and others, "System D--An Integrated Tool for Systems Design, Implementation and Data Base Management," Automated Tools for Information System Design, H.-J. Schneider and A. I. Wasserman, eds., North-Holland Publishing Co., Amsterdam, 1982, p. 67-83.

113. Heckel, Paul, "Designing Translator Software," Datamation, vol. 26, no. 2, February 1980, p. 134-138.

114. Jenkins, C. Wesley, "Application Prototyping: A Case Study," Performance Evaluation Review, vol. 10, no. 1, Spring 1981, p. 21-27.

115. Groner, Gabriel F., and others, "Requirements Analysis in Clincial Research Information Processing -- a Case Study," Computer, vol. 12, no. 9, September 1979, p. 100-108.

116. Hemenway, Kathleen and McCusker, Leo X., "Prototyping and Evaluating a User Interface," Proceedings of the Sixth International Computer Software and Applications Conference, IEEE Computer Society Press, Silver Spring, Maryland, 1982, p. 175-180.

117. Keus, Hans E., "Prototyping: A More Reasonable Approach to System Development," Software Engineering Notes, vol. 7, no. 5, December 1982, p. 94-95.

118. Rocke, Denis, "What is Quality and How is it Maintained?," Proceedings of the Royal Society of London, Vol. 381 A, 8 June 1982, p. 245-261.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93940	2
3.	Department Chairman, Code 59 Department of Administrative Sciences Naval Postgraduate School Monterey, CA 93940	1
4.	Curricular Office, Code 37 Computer Technology Naval Postgraduate School Monterey, CA 93940	1
5.	Professor Gordon C. Howell Department of Information Systems Georgia State University Atlanta, GA 30303	2
6.	CAPT Bradford D. Mercer, USAF Code 52Zi Naval Postgraduate School Monterey, CA 93940	5
7.	Associate Professor Roger D. Evered Code 52Ev Naval Postgraduate School Monterey, CA 93940	1
8.	Assoc. Professor Roger H. Weissenger-Baylon Code 54Wr Naval Postgraduate School Monterey, CA 93940	1
9.	LCDR John R. Hayes, USN Code 54Ht Naval Postgraduate School Monterey, CA93940	1
10.	Mr. Michael R. Kirchner 4343-204 Americana Drive Annandale, VA 22003	3
11.	Professor A. Milton Jenkins Operations and Systems Management Graduate School of Business Indiana University Bloomington, IN 47405	1
12.	Mrs. Mary Jane Kirchner 135 S. Elmwood Ave. Oak Park, IL 60302	1

13. Air Force Contract Management Division 1
AFCMD/KRR
Computer Systems Contract Management Division
Embedded Computer Resources Focal Point
Kirtland AFB, NM 87117

202120

Thesis
K4977
c.1

Kirchner
The software engineering prototype.

4 SEP 90
4 SEP 90
4 SEP 90
4 SEP 90

36978
36978

202120

Thesis
K4977
c.1

Kirchner
The software engineering prototype.

thesK4977
The software engineering prototype.
3 2768 002 10908 4
DUDLEY KNOX LIBRARY