Front-end planning with early test plan scenarios : a strategy to improve systems development in the culturally diverse, technologically complex workplace of the 1990's.

Dorothy J. Eastman
University of Massachusetts Amherst

FRONT-END PLANNING WITH EARLY TEST PLAN SCENARIOS: A STRATEGY TO IMPROVE SYSTEMS DEVELOPMENT IN THE CULTURALLY DIVERSE, TECHNOLOGICALLY COMPLEX WORKPLACE OF THE 1990'S

A Dissertation Presented

by

DOROTHY J. EASTMAN

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 1989

School of Management

FRONT-END PLANNING WITH EARLY TEST PLAN SCENARIOS: A STRATEGY TO IMPROVE SYSTEMS DEVELOPMENT IN THE CULTURALLY DIVERSE, TECHNOLOGICALLY COMPLEX WORKPLACE OF THE 1990'S

A Dissertation Presented

by

DOROTHY J. EASTMAN

Approved as to style and content by:

Van Court Hare Jr

Van Court Hare, Jr., Chairperson of Committee

[illegible]

Susan E. Grady, Member

[illegible]

Conrad A. Wogrin, Member

[illegible]

Michael G. Ketcham, Member

[illegible]

Ronald J. Karren, Acting Graduate Program Director,
School of Management

ACKNOWLEDGMENTS

Many people contributed to this thesis. I would like to thank the chairperson of my committee, Dr. Van Court Hare, for his help and advice throughout my PhD studies, and my committee members, Dr. Susan Grady, Dr. Conrad Wogrin and Dr. Michael Ketcham, for their patience and effort that helped me focus on the important issues in this thesis. I would also like to thank Dr. Tony Butterfield and Dr. Linda Smircich for their continued support--they both were instrumental in the development of this thesis. Several other people contributed insight to the development that underpins this thesis: Marilyn Cross, John Tutt, Bob Frank, Walt Tucker, Bruce Bosworth, Larry Reboulet, Arjun Ayer, Sandy Lavine and Howard Johnson. Special thanks to Dr. George Detlefson, who helped me recognize the most significant result of computerized systems--their social impact.

I would like to give special thanks to my parents John and Genevieve Krawczyk, and friends, particularly Gary and Sylvia Galvan-O'Connor for their encouragement.

I would especially like to thank my family, who helped so much during the last six years, particularly during the years of research when I often spent weeks away from home, and the intense months spent writing this thesis. Thank you Amber, Kevin, Craig and Julie.

Most of all, I thank my husband, Bruce Eastman, who supported my goals, tolerated my ventures, believed in me, and helped me evaluate priorities. Throughout the past six years, he helped us all survive, and grow, as a family. This thesis is dedicated to my family, in acknowledgment and appreciation of their support and dedication.

ABSTRACT

FRONT-END PLANNING WITH EARLY TEST PLAN SCENARIOS:
A STRATEGY TO IMPROVE SYSTEMS DEVELOPMENT
IN THE CULTURALLY DIVERSE, TECHNOLOGICALLY COMPLEX
WORKPLACE OF THE 1990'S

MAY 1989

DOROTHY J. EASTMAN, B.A., UNIVERSITY OF MASSACHUSETTS

Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Van Court Hare

Front-end planning with early test plan scenarios is an effective management strategy for systems development within the more diverse, more complex American workplace of the 1990's. In the 1980's, over seventy-five percent of our system developments failed. Ineffective user-developer collaboration and an undisciplined development environment contributed to this dismal record.

One major challenge is to overcome cultural barriers and develop more effective ways of working together. Within ten years, eighty percent of our new workers will be women, minorities and newly arrived immigrants. In many schools, more than fifty percent of our graduate students in systems development are from other countries.

Another challenge comes from international competition. American corporations try to remain competitive in the world market by increasing productivity and quality. They need new, integrated systems that improve operations as well as process information.

This early test plan scenario thesis develops a strategy and describes a disciplined approach that lets users and developers work together more effectively. In particular, *early test plan scenarios* reduce cultural misunderstanding (rarely addressed in the world of systems) by describing each abstract system task in concrete, physical terms. Scenarios uncover ambiguities, inaccuracies and insufficiencies in system design. Users and developers collaborate to identify operational errors in design and to improve system usefulness.

Even when there are no users, explicit test scenarios reduce misunderstandings between collaborating development teams and among members of the same development team. Finally, scenarios are a powerful tool for ensuring compatibility between integrated systems.

Front-end planning with early test plan scenarios improve systems during the most cost efficient development phase--before coding. Traditional development methods make these improvements during the most expensive phase--after the system becomes operational.

Early test plan scenarios become the development agreement. Clear descriptions of active system operations reduce ambiguity, thereby improving programmer productivity and system quality. Measurable development goals reduce customer-developer conflict.

Early test plan scenarios are an effective systems development strategy for the workplace of the 1990's--they promote a productive, disciplined environment while encouraging people to work together.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

CULTURE, SYSTEMS AND OPERATIONS--THE CHALLENGES OF THE 1990'S

I know a little about nature and hardly anything about men. Einstein

1.1 Statement of Thesis

My thesis: *If* you design a complex system that (1) integrates physical activity with computer technology, (2) requires expertise from several disciplines during the design process, and (3) must satisfy organizational objectives and operational constraints, *then* the system design sequence should place test plan scenarios (described later in this thesis) *early* in the development process, *before* program coding. Traditional development methods test systems *after* program code completion, not *before*.

This early test plan scenario thesis statement may at first appear trite or obvious, but it is not. Its approach incorporates the fundamental strengths of quality control, artificial intelligence, management by objectives, and contract law. As its particular contribution to systems development research, my thesis also includes essential concepts from anthropology and related social sciences.

1.1.1 Multi-Discipline Approach

Quality control strategy continually strives to improve a product by: (1) early recognition of product defects, (2) up front adjustment of production processes, (3) assuring that added value is not wasted on defective products, (4) avoiding expensive rework of the "finished" product, and (5) putting everyone to work to create the best possible product [Mann, 1988, p. 10].

The early use of test plan scenarios lets users identify errors in system design that, if left uncorrected, later interfere with the successful use of the system. Identifying errors *before* system coding follows basic tenets of artificial intelligence: Early pruning of decision trees minimizes wasted effort.

Similarly, both management by objectives and contract law urge parties to an agreement to specify exactly what shall be done, when it shall be completed, and how much it shall cost. These agreements describe *measurable objectives*. Deviations from these objectives can be detected and corrected--or the objectives changed--in an orderly, controlled manner. Once test scenarios have been approved, they become the *measurable objectives* for the development agreement, program coding, acceptance testing, and operational implementation.

1.1.2 Add Cultural Differences

Such reasons alone might be enough to justify the use of early test plan scenarios in the systems development process. It is the purpose of this early test plan scenario thesis to demonstrate other reasons why this approach improves systems development: (1) The need to recognize ambiguous interpretations of functional specifications, (2) The importance of resolving cultural misunderstandings in the design process--before conflicting values and goals are designed into a system, and (3) The value of overcoming cultural barriers that impede user-developer collaboration.

The underlying topic of this early test plan scenario thesis is *social interaction.* This organizational approach--which uses wisdom taken from both technical and social disciplines--offers a new contribution to the systems development field. Early test plan scenarios establish *effective* user-developer collaboration early in the development process, in order to detect and correct operational problems in the system design.

Most extra costs during complex systems developments come from blunders made at the beginning of the development process. Due to major changes in the American workplace, in the next decade these blunders are likely to be more frequent--and even more costly.

1.2 Two Major Changes in the American Workplace of the 1990's

By the turn of the century, American organizations will be faced with two major changes: (1) Workforces will be significantly older and more culturally diverse, and (2) Higher productivity and quality requirements will force the development of a new generation of systems that integrate physical operations with information processing.

This early test plan scenario thesis accommodates these changes. Most traditional development research acknowledges the importance of effective user-developer interaction--but focuses research efforts on improving technologies, tools and design methodologies. This early test plan scenario thesis complements traditional research by focusing on *improving user-developer interaction.*

1.2.1 The Increasing Diversity of Our Workforce in the 1990's

At the end of the next decade, "baby boomers" will still be the largest group in our workforce. By then, they will be in their 40's and 50's. Our workforce has never before been dominated by older workers.

At the same time, there will be a higher demand for new workers. Eighty percent of these new workers will be "women, minorities and immigrants" [Schwartz, 1989, p. 68]. Among these new members of the American workforce, Hispanics and Asians are expected to double their 1986 representation (see Table 1.1 and Table 1.2). Government forecasts predict that by the turn of the century, nearly thirty percent of our new workers will be Hispanic and twelve percent will be Asian [Hymowitz, 1989, p. B1].

Table 1.1 Projected Work-Force Growth Rates.

Group	1976-1986	1986-2000
All women	62.0%	63.2%
Blacks	14.5%	17.4%
Asians	6.9%	11.4%
Hispanics	17.8%	28.7%
White women	43.5%	34.8%
White men	18.1%	8.5%

Source: Bureau of Labor Statistics
[Hymowitz, 1989, p. B1].

Table 1.2 Current and Projected Labor Force Shares by Age, Gender, Race, and Hispanic Origin.

Group	1986	2000
16-24 years old	19.8%	16.3%
55 and older	12.6%	11.1%
Female	44.5%	47.3%
White	86.4%	84.1%
Black	10.8%	11.8%
Asian and Other[a]	2.8%	4.1%
Hispanic Origin[b]	6.9%	10.2%
Median Age	31.5yrs	38.9yrs

a. Includes American Indians, Pacific Islanders, and Alaskan Natives.
b. Hispanics also included in both white and black population groups.
[Leonard, 1989, p. 30]

Moreover, an increasing number of our newest workers have weak job skills (thirty million Americans are illiterate and innumerate). Many newer workers will use English as their second language. Legally and logistically, companies will be less able to require new employees to be fluent in English [Leonard, 1989, p. 30].

Cultural diversity among *systems developers* is increasing. They also come from diverse backgrounds--many use English as *their* second language. In American colleges, among students who are majoring in Management Information Systems, an average of *twenty-five percent* of master's level students, and over *fifty percent* of doctoral level candidates are foreign students (see Table 1.3). A similar situation exists in Computer Science and Engineering disciplines.

User-developer interaction becomes increasingly challenging as users and developers have even more obstacles to overcome to clearly understand *and appreciate* each other's capabilities and requirements.

Table 1.3 Percent of Foreign Students in MIS Programs.

Percent of degree candidates who are foreign students

School	Master's level	Doctoral level
Boston University	30%	70%
University of Arizona	33%	20%
Georgia State University	25%	30%
UCLA	N/A	57%
USC	20%	25%
University of Texas at Austin	22%	60%
Indiana University	25%	30%
University of Minnesota	15%	50%
University of Pittsburgh	20%	60%
MIT	15-30%	56%

[Rifkin, 1989, p. 66]

1.2.2 The Increased Complexity of Integrated Systems

To make matters worse, systems continue to increase in size and complexity. Most systems in the 1980's are of the "first generation". They simply collect, process, store and transfer information. The next, "second generation" of systems--Computer Aided Manufacturing (CAM)--increase speed, quality and flexibility by integrating information processing technology with physical operations. We are just beginning to learn how to merge computer technology with human and mechanical operations [Gold, 1989, p. 139].

Even today, the track record of systems implementation is dismal. For example, a government study found only 2% of developments (see Table 1.4) were fully successful. Recent research suggests that 75% of major developments are failures. They are too costly, are not useful or simply do not work [Foreman and Hess, 1988, p. 65].

Table 1.4 Software Costs Versus Results.

Software Results	Costs	Percent
Paid for but not delivered	$3.2M	47%
Delivered but not used	$2.0M	29%
Abandoned or reworked	$1.3M	19%
Used after rework	$0.2M	3%
Used as delivered	**$0.1M**	**2%**

Reports of nine ($6.8M) federal software projects. The U.S. Government Accounting Office 1979 Report (FGMSD-80-4) [Cox, 1987, p. 4].

In March, 1989, the Department of Defense submitted to Congress a list of twenty-two technologies that it considers critical to the long term superiority of the U.S. weapons systems. *Improving software producibility* ranked third on this list [Norman, 1989, p. 1543].

Systems development is difficult now. Integrated systems of the 1990's will intensify this difficulty. Unless we are able to develop better ways of working with each other, the increasing diversity of our workforce will compound the risks of this already treacherous process.

1.3 Culture, the Underlying Theme of this Thesis

Culture is the underlying theme which holds this early test plan scenario thesis together. Although the content of this thesis supports techniques that improve the quality and productivity of a development, this early test plan scenario thesis is important because it improves the interaction between users and developers--two groups that often have divergent cultures.

1.3.1 Cultural Misunderstanding

By definition, people who share the same culture have a common storehouse of memories, experiences and traditions. They speak the same language. They comfortably share similar values and goals. Ambiguity vanishes as they share mutual understanding. They are "at home" with each other.

Persons who have acquired different cultures may misinterpret each other's intentions. We see through the lens of our own culture and hear what our culture teaches us to hear. The more unsettled we are in each other's presence, the more apt we are to "mis-hear" each other.

Cultural differences elicit emotional responses which may impair hearing and block understanding [Murphy, 1988, p. 43]. Strangers can share (or be easily taught) mutual understanding of concrete objects, such as a "table" or "chair." Disagreement emerges from differences in interpretation of abstract terms. "Good" and "fair" have different meanings in different cultures. An abstract term in one culture, may trigger unanticipated, and unappreciated, responses in another culture.

1.3.2 Workforce Cultures

In particular, workforce cultures can conflict. Accountants see the world in terms of rows and columns, managers view it in terms of strategies and policies. Electrical engineers see the world electrically, mechanical engineers view it mechanically.

Programmers look through an *abstract* cultural lens and see a fantasy world of logic and analogies. Users look through a *physical* cultural lens and see a concrete world of things and actions. Differences in workforce cultures have always caused problems during developments because developers have difficulty "understanding the users' needs."

Even if no users are involved, the problem persists. Customers who pay for systems may assume that developers fully understand their (the customers') goals and values, *even if these goals and values are never explicitly defined.*

Networking, telecommunication and systems integration give rise to a further obstacle. Teams of developers, often members of different organizations (and frequently located thousands of miles apart) must build systems that work together. Cultural misunderstandings intensify the difficulties of working separately to build systems that must eventually join and work together.

Finally, cultural misunderstandings may divide members of a development team. Systems people are not a standardized group--they are themselves of different cultures (refer back to Table 1.3). Conflict among team members devastates a systems development project.

1.3.3 Culture in the Development Setting

Accordingly, developers and users who sit together to design systems are often dissimilar. When cultural differences interfere with user-developer collaboration, the final system often fails because it is not useable within its intended environment.

1.4 The Test Scenario

Organizations need a tool that will permit clear and timely communication in complex situations characterized by cultural diversity. That tool is the early test plan scenario of this thesis. A detailed description of early test plan scenarios and their use appears in the next two chapters. Chapter 4 through Chapter 6 describe how early test plan scenarios fit into the overall System Development Life Cycle and into systems development research.

The *effect* of early test plan scenarios is to reduce the number and variety of misunderstandings. Early test plan scenarios convert abstract concepts into concrete, measurable actions. The *result* of early test plan scenarios is an improved likelihood that the system will be used and will have positive benefits.

The use of early test plan scenarios assures the operational acceptability of the system design, forestalls design blunders, and provides a clear and measurable work specification. In short, early test plan scenarios stabilize the system development work process.

1.4.1 Creating the Test Scenario

Briefly, to create a test scenario, we define the basic sequence of operations within the system's intended environment. In the illustrative case used in Chapter 3 (the tutorial within this thesis), the operational sequences are: (1) Hours, (2) Half-shifts (four hours), (3) Shifts (eight hours), (4) Days, and (5) Weeks. We cite every task the system must perform during each sequence. Each task is then broken down into a series of concrete action test units. The process of creating and using scenarios incorporates four steps (see Table 1.5).

Table 1.5 Four Steps in Test Scenarios.

1. Describe Action	- What, when, where and how people, machines and systems will physically act.
2. Define Results	- What the developers' think *should* happen.
3. Evaluate	- Group discussion among users and developers to predict what *actually will* happen.
4. Improve	- Correct and modify system design to make it more useful.

1.4.2 Why Early Test Plan Senarios Work

The Software Engineering Institute (formed at Carnegie Mellon University by the Defense Department) researches systems developments throughout America. They have found three indicators that separate the two percent of developments which succeed from the ninety-eight percent which fail: (1) Clear system definition, (2) Quantified process measurement, and (3) Systems process control [Humphrey, 1988, p. 73].

Early test plan scenarios work because they support a clearly defined, disciplined development environment. Please consider the two essential characteristics of early test plan scenarios that define, measure and control the system development process:

(1) Clear goals and corrective mechanisms described in operational terms.

(2) Early resolution of potential conflicts and blunders.

1.4.3 Guideline for Use: Adapt

Discipline improves the development process. But care must be taken when implementing methods that increase discipline. If methods that work in one development are *adopted*, without change, into a different development environment, *discipline* may quickly become *regimentation* [Humphrey, 1988, p. 78]. To preserve flexibility and creativity and yet maintain discipline, developers must *adapt* methods to fit the *unique character* of a new development situation.

This early test plan scenarios thesis is a *guideline*. It offers *ideas* that can help projects succeed. The detail incorporated within the tutorial (Chapter 3), may initially overwhelm the reader. This tutorial is similar to a programmer's manual. It contains ideas that have improved actual complex developments--but it also includes extensive detail for completeness.

Common sense dictates that developers select and adapt *only* those portions of this tutorial that *improve* a particular development situation. Otherwise, the early test plan scenario method--that could promotes disciplined creativity--becomes a regimented, unwieldly trap.

1.4.4 Special Advantages of Early Test Plan Scenarios

Please note the many advantages of the early test plan scenario approach when it is applied before program coding begins:

1.4.4.1 Management Advantage

From a management perspective, the set of verified test plan scenarios provides the basis for *management by objectives.* Individual test scenarios become *mini-contracts*, which are then combined into an overall development contract. See Chapter 7 for a further discussion of this management and legal advantage.

1.4.4.2 Users Contribute

When users participate in early test plan scenario evaluation, they develop an in-depth understanding of the system that reduces training requirements. Users who understand a system make fuller use of a system's inherent capabilities.

Technological innovation is continuous. When users understand their systems, they are more likely to suggest enhancements that increase the usefulness of future versions of the system.

1.4.4.3 Developers Produce

Early test plan scenarios help people *develop and meet* schedules--they stabilize the work process. An initial review of this early test plan scenario thesis might lead one to assume that the time required for analysis and documentation, for developing test scenarios, and for scenario review, would *decrease* developer productivity. In practice, this approach significantly *increases* productivity.

To illustrate, (see Table 1.6, p. 15) one development team used a version of early test plan scenarios and increased its productivity from the national average of seven lines per day to twenty-nine lines per day--*a 300 percent increase in productivity!* Engineers delivered a substantially better product (i.e. with fewer customer-discovered errors) in a significantly shorter time [Poston and Bruen, 1987, p.60].

1.4.4.4 Continuity Preserved

A single system's development may span several years. Managers, developers, and users, who originally defined operational requirements often leave the development team. If the new team members form faulty interpretations of operational requirements, disruptions and errors increase. The concrete descriptions of system activity in early test plan scenarios *preserve continuity* of operational intent.

1.4.4.5 Quality Improves

Early test plan scenarios improve system quality by reducing failure density (refer again to Table 1.6, p. 15). Failure density measures the number of customer-identified errors per thousand lines of delivered code. When early test plan scenarios are reviewed, users identify operational errors in system design. The design is corrected *before* programmers concretize the design into system code.

Table 1.6 PIMS Project* and Industry Comparison.

Characteristic	Measure	PIMS Project	Industry-reported
Size	Pages of documentation	2,680	--
	Delivered lines of code	27,719	--
	Number of test cases	1,056	--
Complexity	Real-time process control	average	--
Quality	Development failure data		
	Total failures	12	--
	Failure density**	0.43	5-30
	Testing comprehensiveness***	1	
	Customer failure data		
	Total failures (first year)	2	--
	Failure density**	0.072	1.3
Cost	Staff-hours	7,627	--
Productivity	Deliverable lines of code per staff-day	29	7
Schedule	Calendar months	9	

* *Process Information Management Subsystem (PIMS) Trending Project at Leeds&Northrup Systems in North Wales, PA, 1986*
** *Failure density=number of failures per thousand lines of code.*
*****If comprehensiveness is unknown, the development-based failure density number has no meaningful reference point (for example, a low failure density could be due to poor testing).*
[Poston and Bruen, 1987, p. 60]

1.4.4.6 Acceptance Assured

Early test plan scenarios act to assure acceptance of the system. Users, sponsors and developers *understand, evaluate, improve* and *approve* the acceptability of a system's design. Test plan scenarios become the *measurable development agreement.*

1.4.4.7 Rework Reduced

When faulty design commitments are made, they propogate. Early mistakes solidify. We are left with hard, frozen designs, extensive program code and escalating personal and financial investments. When users eventually identify the errors, the system is often dropped.

Early test plan scenarios reduce rework by uncovering design blunders *before* programmers transform the design into code. Early test plan scenarios permit productive process control: (1) Discover the blunder, (2) Improve the design, and (3) Code the design.

1.4.4.8 Total Costs Reduced

The most expensive phase of systems development begins *after* developers have "completed" their work and the system becomes operational. System "rework" to correct design blunders (discovered during system operations) are expensive. A modification required *after* a system begins operations may cost *one hundred times* as much as that same modification made *before* system coding. Over the entire life cycle of a system, *the cost of rework normally exceeds the total cost of the entire pre-operational development* [Carroll, 1989, p. 1].

Early test plan scenarios reduce rework, thereby reducing the total cost of modifications. Thus, any extra cost or delay caused by early test plan scenarios will be repaid--with dividends--over the entire life of the systems development life cycle.

1.4.4.9 Scenarios Recycled

Developers can catalogue and recycle early test plan scenarios. Libraries of scenarios improve subsequent developments. Novice users shorten learning curves by examining recycled scenarios. New customers review recycled test plan scenarios to evaluate and select *potential* operational attributes *before* the actual specification process.

Developers reduce costs by modifying (rather than recreating) recycled test plan scenarios. When test plan scenarios are explicitly linked to functional and design documents (as described in the next two chapters), these documents are more easily recycled. Linking *test plan scenario modules* to *program modules* further improves productivity.

1.4.4.10 Marketing Opportunity

Catalogued test plan scenarios (as described in the previous section) are valuable marketing resources. Building contractors use photographs of houses to market their services. Similarly, systems developers, marketing their services to potential customers, can use recycled test plan scenarios to illustrate the value of their product.

1.5 Early Test Plan Scenarios Complement Other Methodologies

Early test plan scenarios complement other methodologies. Each development situation is unique. Different developers prefer different techniques. Generation and evaluation of early test plan scenarios, before system coding, improve quality and productivity, *regardless* of which other development techniques are used.

1.5.1 Comparing Early Test Plan Scenarios to Prototyping

Prototyping increases user involvement early in the *program* development cycle. This early test plan scenario thesis expands the prototyping philosophy to incorporate user expertise early in the *complex systems* development cycle.

In other words, *prototyping* improves the interaction between users and developers who are designing *programs* that process information. *Early test plan scenarios* improve the interaction between developers and users who are designing *complex systems* that integrate information processing with physical operations.

1.5.2 Comparing Early Test Plan Scenarios to Modular Top-Down Design

Modular top-down design encourages the development of reliable and flexible system structures by building a solid foundation before adding intricate details. When top-down design is rigorously enforced, the final system has a quality that permits gracious modifications.

The early use of test plan scenarios allows the system's fundamental design to be modified at an early stage--during a time in which the restructuring of its modular top-down design can be carefully planned and monitored.

Unfortunately, the most damaging errors discovered by customers are not errors in *program logic* but errors in *operational impact*. Post-operational system modifications, aimed at fixing these errors, can sabotage the original design. The early use of test plan scenarios preserves the fundamental goodness of modular top-down design by reducing the number and severity of post-operational modifications.

1.6 Test Plan Scenarios--the Missing Link in Complex Developments

When test plan scenarios are developed and reviewed prior to system coding, they provide an early link between users and developers that is missing from traditional development approaches. Early test plan scenarios overcome cultural obstacles to user-developer interaction. User expertise is incorporated early into the system design during the critical, pre-coding phases of complex developments. These scenarios provide structure and focus that improve developer productivity and system quality.

1.7 A Personal Note

I have had the opportunity to be part of many different cultures. As a child, I grew up in an multi-lingual environment in which English, Polish and Russian were spoken. I subsequently studied, and used, several other languages. As a teenager, I worked with Dr. Martin Luther King and Rev. Jim Robinson, founder of Crossroads Africa.

My doctoral reseach involved nearly three years of participatory observation in a subsidiary plant of a major corporation, during the development of two, multi-million dollar, integrated systems. The plant is located in an inner city. Ninety-five percent of the plant's workers and management are minorities, primarily Blacks and Hispanics. The millions of dollars allocated to these two developments were meant to increase the quality and productivity of this plant's operations.

The first development (for a Storage/Delivery system) was initially a "failure." Users did not participate in the design. After fifteen staff-years of effort and $15 million (a 300% escalation in budgeted time and cost) there was no useable system.

This project became successful only after the plant staff appointed a new project manager. Users became involved in a re-design of the system. We wrote extensive test plan scenarios for users and developers to evaluate. The evaluations helped to improve the system's design. Scenarios were revised to reflect these improvements. When the final scenarios were approved, they became a clear guide for all subsequent programming. These test scenarios became acceptance tests, the measurable objectives of the development agreement. With minor revisions, the test plans scenarios were converted into user manuals.

Another group within the plant developed a Receiving/Order system to link production with the warehouse. The approach presented in this thesis was throughout this development. The final result was a useful system, developed within time and cost expectations. Both systems are still in use.

During the course of this research, I gathered and catalogued several thousand pages of documents, notes, communications, letters, interviews and observations.

An unanticipated bonus is that users who were involved in this development were challenged and motivated by their participation. They have become an enthusiastic group who are dramatically increasing their skills, aspirations and contributions to their plant and to each other.

It was successes of this type that motivated me to propose the early test plan scenario thesis.

CHAPTER 2

EARLY TEST PLAN SCENARIOS SUPPORT FRONT-END SYSTEMS PLANNING

It must be remembered that there is nothing more difficult to plan, more doubtful of success, nor more dangerous to manage, than the creation of a new system. For the initiator has the enmity of all who would profit by preservation of the old institutions and merely lukewarm defenders in those who would gain by the new ones.
Machiavelli

2.1 Front-End Planning

Front-end planning (planning which occurs before coding) manages the course of a systems development by planning *before* coding. Front-end planning is basic to successful development of integrated systems (which integrate information processing with operations). Research sponsored by the Department of Defense studied several hundred software development organizations. Only *two percent* of these organizations attained even a medium level of quality, when quality was measured by developing *useable* systems, *on time*, and *within estimated costs*. Of the two percent of developers who produced quality systems, *all used extensive front-end planning*.

Traditionally, systems developers resist planning. Developers who work on integrated systems (as described in the preceding chapter) no longer have the luxury of using individual approaches to systems development. The need has become too critical for continued system development failures [Norman, 1988, p. 1543].

2.2 System Force

When a complex system becomes active within an organization, it is no longer simply a set of programs. It becomes a dynamic force. An active system is a conduit for communication. It is a filter and modifier of information. An active system changes an organization's operational environment by altering the interaction of its people, machines and other systems. Without thorough front-end planning, *programmers who code the system determine its operational impact.*

2.3 Successful Management Brings Users and Developers Together

Successful systems development management ensures that questions which determine a system's useability are asked at the right time--and that people who have the relevant expertise are allowed to answer these questions. Often in systems development, people with the most relevant expertise, the users, do not evaluate the usefulness of a system *until it is activated within the organization.*

Successful systems development management does not simply help developers "determine the users needs." Successful management lets users and developers work together at the drawing table [Schrage, 1989, p. 10]. Early test plan scenarios are a powerful management tool.

> In the typical jargon of software developers, "testing" means "proving" a design. Usability testing, on the other hand, is done to find out what is wrong with a design. This shift in purpose necessarily causes a change in attitude toward users and the product being tested [Potosnak, November 1988, p. 83].

2.4 Management of Front-End Planning

This early test plan scenario thesis presents a front-end planning approach which incorporates early test plan scenarios. Front-end planning includes four phases: (1) Describing objectives, (2) Analyzing functionality, (3) Defining structure, and (4) Evaluating organizational impact. Using the feedback tactics of quality assurance, each phase includes five tasks: (1) Plan, (2) Produce, (3) Test, (4) Improve, and (5) Approve. Each phase produces documents which are specifically designed to support subsequent phases. Figure 2.1 describes these phases and the tasks associated with each phase.

2.5 Four Planning Views

Each planning phase uses a different view of the planned system. The *objective view* defines the purpose of the system, the reasons why an organization plans to allocate resources to this effort. The *functional view* describes the components of the system, exactly what activities it must be capable of accomplishing. The *structural view* defines how all components will fit together to form a stable system.

The dynamic scenarios of the *operational view* demonstrate how the structured components should act within the organization's environment. Will the system, as it is currently specified, be useful and useable within the temporal, physical and sequential conditions of the actual organization?

	TASK 1 PLAN	TASK 2 DO	TASK 3 TEST	TASK 4 IMPROVE	TASK 5 COMPLETE
OBJECTIVE	Identify problems. Specify Objectives, Resources, Time & Cost Limitations A.1.1	Write D.I.1 Objectives Documents, defining objectives, benefits, costs. A.1.2	Trouble-Shoot Objectives. Will the system be useful & useable? A.1.3	Brainstorm. Challenge, propose, evaluate, & accept improvement A.1.4	Sponsors Sign-off Objectives Document. A.1.5
		<--	<--	<--	<--
FUNCTION	Analyze functional processes, data, logic hardware, needed to meet system objectives required. A.2.1	Write Functional Analysis. Describe Processes, algorithms, data,soft/ hardware Requirements A.2.2	Walkthrough Functional Analysis. Is it clear unambiguous consistent accurate & complete? A.2.3	Improve analysis. Simplify, reduce cost make more productive. A.2.4	Users and sponsors sign-off Functional Analysis Documents. A.2.5
		<--	<--	<--	<--
STRUCTURE & DESIGN	Analyze structure. Design a logical, complete system to satisfy all functional needs. A.3.1	Produce Structured Design. Use tables diagrams, flowcharts pseudo code Link each element to function. A.3.2	Prove Structured Analyses Will they correctly completely support all parts of Functional Analysis Documents? A.3.3	Challenge & improve analysis. What parts of design are missing incomplete inaccurate too complex redundant? A.3.4	Developers sign-off Structured Design Documents. A.3.5
		<--	<--	<--	<--
TEST PLAN SCENARIOS	Activate analysis. Describe physcial interaction of system with people machines & all systems in actual operational environment A.4.1	Write Operational Test Plan Scenarios using all sequential spatial, temporal conditions of normal & exceptional operations. A.4.2	Team walk-through of scenarios. Personal review by users of portions of test which impact their own organizational role A.4.3	Rigorous exercise of scenarios. Brainstorm to discover flaws. What can improve usefulness, useability, of system in organization? A.4.4	All sign-off Test Scenarios Users: the System will be useable. Sponsors: System will be useful. Developers: System can be built. A.4.5 To PRODUCE

Figure 2.1 Front-End Planning With Early Test Plan Scenarios.

2.6 Each Planning Phase Asks Different Questions

Each phase supports different questions. The tutorial (presented in the next chapter) suggests guidelines for asking the most useful and appropriate questions during each phase. The answers are presented in documents, using formats selected to facilitate communication (see Table 2.1). These formats are designed to help users and developers understand, challenge and modify plans during the early stages of systems development. Each component (of each phase's documents) is formally linked to corresponding components in documents of the other phases. Rigorous linkage decreases ambiguity while increasing traceability and consistency in the design.

Table 2.1 Front-End Planning's Questions, Documents and Formats.

MAJOR QUESTIONS	DOCUMENT FOR ANSWERS		FORMAT
I. How should the new system affect the organization?	D.I	System Objectives Documents	Narrative
II. What should the parts of the system do?	D.II	Functional Requirements Documents	Text, Tables, Algorithms
III. How should the parts of the system fit with each other & with the organization to form a stable system?	D.III	Structured Analysis Documents	Blocks, Charts Diagrams
IV. How should the system interact with the people machines & systems of the operational environment?	D.IV	Test Plan Scenarios of Operations	Script, "Screen Plays"

Successful systems development integrates these four views into a single, cohesive system. The management of systems development, and particularly the management of the requirement specification process, is difficult because the language, evaluation criteria, communication methods and cognitive understanding tools are different for each view.

A crucial factor in the development process is the ability of the participants to challenge system plans. Without a firm structure for questioning and improving problems in the initial plans, these problems often remain undetected. When problems surface during acceptance testing or operations, they are often so costly to correct that the system itself is scrapped. *These four phases ensure that each view receives sufficient attention and that people who best understand each view are able to contribute their unique expertise* (see Table 2.2).

Table 2.2 Contribution of Four Planning Phases.

1. Recognizes the unique importance of each of four views: Objective, Functional, Structural, Operational.
2. Establishes questioning as a fundamental process in systems development, clarifying ambiguities and promoting improvements.
3. Manages a collaborative structure which enables people with different skills, responsibilities and understandings to work together, complementing each other's efforts.
4. Selects document formats which increase understanding of views.
5. Enforces a rigorous linkage system which ensures that the products of different views form a cohesive entity.

2.7 Early Test Plan Scenarios Integrate Different Views

Early test plan scenarios are scripts that integrate the three preceding views into a unified whole. Test plan scenarios exploit the expertise of users by enabling them to imagine the active system and to infer the operational effects of the system. Scenarios describe *each action* of the system, *and each expected response* of the people, machines and other systems that interact with the system.

Scenarios are screenplays. They include all of the prerequisites for the actual test, detail the content, location and condition of all required data, material, machinery and persons. Scenarios clearly describe all relevant activities, movements and changes, *using the actual sequence in which they occur.* Scenarios explicitly describe the exact response, or the acceptable range of responses, to each of the system's actions.

Scenarios present events using a clear, simple format. Scenarios describe expected responses in quantified, measurable terms. The evaluators of the test scenarios assess the usefulness, *within the organization,* of the characteristics, effects, constraints and limitations of the planned system. Scripts let users improve the system's usefulness by inferring the *actual operationalization* of the system.

Operational test plan scenarios help users *objectively* evaluate a proposed system's useability. When the user is asked to evaluate a system, or a portion of a system, which is already operational, the user merely *approves* a *design* [Potosnak, November 1988].

When users evaluate early test plan scenarios, they evaluate, modify and *improve* the design itself, using her/his expertise in the actual operational environment.

We have little understanding of those processes by which people perceive and infer. Researchers of cognitive perception express their "humble appreciation of the enormity of our ignorance about how these mechanisms work" [Wilding, 1983, p. 277]. We do know that the script format is a traditional and powerful method for people to communicate plans and concepts concerning proposed activities.

For thousands of years, people have used scripts to describe interaction in screenplays. Script integrates separate entities, (whether people, machines or systems), which have different motives, plans and possibilities. Scripts allow and *direct* the focus of attention to be directed towards the *interaction* of separate people, structures and things.

To be useful, a system must *improve* the interaction of people, machines and systems within an organization's operational environment [Lucas, 1975]. Most forms of specification are capable of addressing only the isolated, internal and individual characteristics of a program. Some methods are able to document the flow of information among programs and between programs and the "outside world".

These formats for documentation are essential for defining the internal characteristics and the interface communication channels of systems. Scenarios do not replace formal specifications, tables, charts and diagrams. *Scenarios illustrate how a system, defined and specified with these traditional system development tools, will dynamically interact within an organization.*

> Detailed scenarios, which provide more elaborate, and thus more expensive, operational descriptions, are even more effective than simple scenarios in clarifying a system's human engineering aspects [Boehm, 1984a, p. 81].

When a system is considered to be an active entity within a dynamic organization, system developers become playwrights who establish the rules of interaction among people, things and systems. Scripts force a director of a play to exactly follow an author's plans.

Once early test plan scenarios are formally approved, programmers must follow the directions of the system planners [Poston and Bruen, 1987, p. 59]. When difficulties arise during coding, programmers collaborate with users and sponsors to resolve conflicts between code and concepts.

Early test plan scenarios become the acceptance tests for evaluating the successful completion of the system. When early test plan scenarios are formally approved, prior to system coding, they establish quantified and unambiguous measurements for evaluating the system itself.

2.8 System Planning--Burden or Benefit?

System developers often argue that detailed planning is neither necessary nor possible. Planning is seen as interfering with the creative process and limiting the potential goodness of the final system. A variety of "reasonable" excuses are found to skip quickly through the planning stage and plunging into the exciting, challenging, and more satisfying phase of system coding. Programmers offer many excuses: "Users do not know what they want", "Until you begin work you cannot identify what should be done", and "Planning delays the real work". When programmers bypass the planning stage, the puzzles are greater, more interesting and more fun to solve--*and the price of systems development increases.*

> Programming systems can, of course, be built without plans....But since there is no general theory of the whole system, the system itself can be only a more or less chaotic aggregate of subsystems whose influence on one another's behavior is discoverable only piecemeal and by experiment. [Weizenbaum, 1976, p. 119].

CHAPTER 3

A FRONT-END PLANNING TUTORIAL THAT HIGHLIGHTS EARLY TEST PLAN SCENARIOS

Invention, it must be humbly admitted, does not consist in creating out of void, but out of chaos; the materials must, in the first place, be afforded: it can give form to dark, shapeless substances, but cannot bring into being the substance itself. Mary Shelley, *Frankenstein.*

3.1 Two Integrated Developments Underpin Tutorial

A major, multi-national corporation sponsored the two, multi-million dollar system developments that underpin this tutorial. Both developments produced integrated systems (systems that merge information processing with operational control as discussed in Chapter 1). The developments spanned several years during the mid-1980's.

3.1.1 Systems Designed to Improve Productivity and Quality

Both systems were designed to increase the productivity and quality of the manufacturing processes within an urban subsidiary of this corporation. The first system supports automated production control: (1) Inventory management, (2) Job control, and (3) Production order control. The second system supports automated material storage and delivery: (1) Rotary Rack Storage, (2) Empty tote control, and (3) Automated guided vehicle delivery.

3.1.2 Extensive Development Research

Several thousand pages of documents, letters, memoes, meeting notes, interviews, and detailed, daily summaries were gathered during nearly three years of full-time participant observation with these two developments. This research occurred both at the plant (located in a New England city) and on-site in California (location of the system development organization responsible for the major material handling and controls software).

All of the documents are catalogued. A detailed case study was written, focusing on the interaction of the participants during these three years of systems development. Additional site visits and interviews were made during the final writing of this thesis.

3.1.3 A Gentle Warning: Use Common Sense and Creativity

The potential user of this tutorial is urged to use her/his common sense and creativity to *select and adapt* components. Extensive detail is included in this tutorial to thoroughly explain how decisions were made and documents designed--*not to define every step which the user of this tutorial must follow.* Structured flow diagrams are included to illustrate the relationship between human activity, decisions and documents during the course of this front-end planning approach--not to concretize the tasks.

Methods which support disciplined development environments promote regimentation if they are simply lifted from one project and adopted by another [Humphrey, 1989, p. 78]. Each development situation is unique--

approaches which succeed in one situation must be adapted to meet the unique requirements of another situation. Regimentation caused by simply adopting a methodology stifles creativity.

This tutorial uses specific tools, such as SADT^{tm}. These tools and techniques are used to demonstrate the planning requirements of each phase. Each development process is unique--each developer has her/his own favorite sets of tools. The use of SADT, or any other specific approach, is not meant to imply that these are the *best* tools. These tools are included because they were successful during the two developments researched for this thesis.

The ideas represented within this tutorial emerged from the extensive effort, creative abilities, and prolonged (and intense) interaction of many people. These ideas helped a diverse group of people, situated on each side of America, work together to successfully build a *useful* integrated system.

3.2 Early Test Plan Scenarios--Highlight of Front-End Planning

This tutorial offers guidelines for the front-end planning of four development phases. Each phase represents a different view of the system being planned. This tutorial discusses the *process* of front-end planning associated with each view.

Views are presented in the same sequence in which they are accomplished: Objective, Functional, Structural, and Operational. The highlight of this thesis is the fourth, operational view that develops early test plan scenarios.

3.3 Phase One: Viewing System Objectives

Front-end planning begins by defining an organization's objectives, asking how a system could *improve* the organization. Objectives may be described simply, in a single document, or require a complex, multi-volume set of documents.

The objective view identifies the impact a system will have on an organization (see Figure 3.1). Objective Documents describe current operations, evaluate problems and needs, and define the benefits of the system. Five tasks help develop objectives (see Table 3.1).

Table 3.1. Five Tasks to Develop Objectives.

(1) Plan Objectives - Identify problems for the system to address. Define specific goals for the system to meet.
(2) Describe Objectives - Write the System Objectives Document, D.A.1. Describe problems to solve, goals,benefits.
(3) Test Objectives - Trouble-shoot the Objectives Document.
(4) Improve Objectives - Brainstorm, propose and improve.
(5) Approve Objectives - Sign-off Objectives Document, D.A.1.

3.3.1 What Are the System Objectives?

Section D.A.1.1 of the Objectives Document states the overall objective of the system within the organization--and the proposed methods for reaching this objective.

D.A.1.1
The task of establishing a comprehensive warehouse system for ABC Corporation is to bring about the capability of carrying a minimum amount of inventory and the distribution of material to product lines in orderly fashion. This goal will be met through implementation of a totally integrated system comprised of a variety of automated mechanical equipment controlled by the software to be developed.

```
                 __________   _____________   ______________
                |          | |             | |              |
                | A. PLAN  ----> B.PRODUCE  ---->C. MAINTAIN |
                |__________| |_____________| |______________|

 _____________   ______________   ________________   ______________
|             | |              | |                | |              |
|A.1 IDENTIFY ---> A.2 DEFINE  ---> A.3 STRUCTURED ---> A.4 TEST PLAN|
|  OBJECTIVES | | FUNCTIONALITY| |    ANALYSIS    | |   SCENARIOS  |
|_____________| |__________/\__| |________________| |______________|
                           ::
                           .......................
    Organization's         : D.I.I.  System      :
     Operational           : Objectives Documents :
     Environment           :..................../\.:
         ::                                       ::
      ___::_______________                  _____::_____________
     |   \/               |                |                    |
     | Sponsors identify  |                | Sponsors signoff   |
     | system objectives: |                | Objectives Document|
     | problems to solve, |                | D.I.1.,  agreeing  |
     | personnel, cost,   |                | to accept system   |
     | time and equipment |                | when it satisfies  |
     | constraints.       |                | all objectives.    |
     |                    |                |                    |
     | A.1.1 PLAN SYSTEM  |                | A.1.5 APPROVE      |
     |        OBJECTIVES  |                |    OBJECTIVES      |
     |___   ______________|                |_______________/\__|
         ::                                                ::
         ::                                                ::
 ________::__________   ____________________   ___________::________
|        \/          | |                    | |                     |
| Write Objectives   | | Trouble-shoot      | | Brainstorm.         |
| Document, I.1.1.   | | Objectives.  Will  | | Propose & evaluate  |
| With text, describe| | any system which   | | new ideas. Promote  |
| problems to solve, | | meets objectives,  | | responsible changes |
| system goals and   | | help organization  | | which improve       |
| benefits, resource | | solve its problems | | objectives.         |
| & cost limitations.| | & achieve its goals| |                     |
|                    | |                    | |                     |
| A.1.2 DESCRIBE     | | A.1.3. TEST        | | A.1.4  IMPROVE      |
|    OBJECTIVES    :::::> OBJECTIVES      :::::>    OBJECTIVES      |
|____________________| |____________________| |_____________________|
```

A.1 IDENTIFY OBJECTIVES

Figure 3.1 Identify Objectives A.1.

3.3.1.1 How Does the Organization Operate Now?

Section D.A.1.2 of the Objective Document summarizes current operations within the organization. This section describes its current activity using text to highlight specific problems and inadequacies in the operational environment.

D.A.1.2
This section is an overview of current material processing methods used at the Main Street location of the ABC Corporation. The operations described are conducted in Building 10.

3.3.1.2 Which Current Systems Will the New System Affect?

One section of the document is reserved for each active system within the organization that will be affected by the new system. An understanding of all systems currently active within the operating environment is needed to plan a new system which will replace, modify or interact with other systems within the organization.

D.A.1.2.1
Receiving - Material is received from various vendors at the receiving docks and a visual audit is made using the freight bill.

D.A.1.2.2
Putaway - Materials are put away in high bay storage and entries are made in the computer based locator system.

3.3.1.3 How Will the System Help the Organization?

Section D.A.1.3 summarizes the system's benefits and defines the system's impact on the organization. The addition of new benefits or the elimination of current problems should be specified. Before the specification process begins, sponsors should have an explicit list of benefits which they expect the organization will receive from the system. These benefits may involve hardware, software and processes.

In many cases, developers, particularly if outside of the sponsoring organization, will not yet be working under a development contract, but under a limited, pre-development specification contract.

D.A.1.3
The proposed system, for ABC Corporation, will permit scheduled Just-In-Time delivery of parts to manufacturing.

D.A.1.3.1 Hardware:
- Self-guided, Self-propelled Delivery Carts to reduce delivery time from a current average of 30 minutes to a maximum of 20 minutes.

D.A.1.3.2 Software:
- Real time control of carts using micro-processers to increase responsiveness of delivery while reducing delivery personnel by 20%.

3.3.2 What Resources Will the Organization Allocate?

What personnel, dollars, and equipment resources are available during the design, development, implementation, modification and operation of the final system? Resource guidelines should include expenditure boundaries.

D.A.1.4 Costs
- This project is expected to conform to the expenditure limits of the organization's second level of priority projects.

3.3.3 When Should the System Be Operational?

It is premature during this initial stage to assign specific implementation dates. Forcing a development to meet inappropriate schedule demands disrupts the production process, increasing its risks and consequent delays. It is important to establish *time boundaries* which represent actual organizational needs, and to assign levels of importance to each boundary.

During development, decisions will be made which balance the time required to develop, test or implement a particular function with the expected benefit of the function. Realistic and prioritized time boundaries assist in determining which functionality to develop and which to eliminate, modify or postpone. In some situations, the schedule is critical and functionality will have to be sacrificed. Othertimes, close adherence to a schedule is less critical than full development of functionality.

In a development environment in which sponsors, users and producers are communicating openly and are mutually directing their efforts towards the successful completion of the system, issues of scheduling, functionality and associated costs are evaluated objectively. When conflicts develop between groups, issues of schedule and functionality may erupt as a major impediment to development success.

D.A.1.5
On July 1, 1990, the ABC Corporation must begin production of a new product line. It is essential that the system be operational before this date, even is some functionality must be ommitted.

3.3.4 Who Will Develop the System?

Section D.A.1.6 defines who will develop the system. Organizations have several choices when selecting developers. Will the system be developed in-house, or will outside vendors be hired? Will a turn-key system be purchased which is developed by a software house totally independent from the sponsoring organization? Who will participate in the development process? Will workers, managers, MIS persons, engineers, executives be involved? To what extent is participation of each group expected during development?

D.A.1.6 The basic system will be developed by a consulting firm. ABC corporation will assign a team of three warehouse supervisors and three warehouse users to participate, on a full time basis. These six people, under the direction of the warehouse manager, will be full members of the development team, working closely with systems developers from the consulting firm.

3.3.5 How Will the Development of the System Be Managed?

Most management strategies assume that teamwork by sponsors, users and producers is required for successful system development. Once a project manager is selected, it is the manager who determines the management philosophy of the development. When developments flounder, new management is often assigned to the project, and the project management techniques are changed. The disruption caused by a change in personnel is aggravated by a change in managerial control. Evaluation of system management considerations, prior to the development itself, reduces some of these risks.

To promote a controlled development, the membership, authority and communication routes of the development team should be determined at the onset of the development. The team's composition should reflect the organization's management philosophy and be appropriate for the nature of the system and the conditions of the development environment.

D.A.1.7 The project team will use Collaborative Management. The project management team will consist of three users, two sponsors and one producer. All decisions will be made by majority vote of the team. In the case of a stalemate, the project manager, to be assigned from the user group, will cast the deciding vote.

3.3.6 How Will the Development Be Phased?

Strategies concerning phases alter a development's priorities and risks. Three basic ways to phase a development are monolithic, incremental and evolutionary [Melichar, 1980).

A monolithic approach establishes all functional requirements, to a detailed level of analysis, for all system functions, prior to any system development. This method works well if the system requirements are known and firmly bounded, if the operating environment is stable, and if the hardware and software used are proven and reliable.

Incremental development establishes overall system requirements prior to development but recognizes that the completion and implementation of one set of functional processes may change the operating environment in unanticipated ways. The use of automated delivery carts may suggest changes to the anticipated system. The operational environment may change during the development. Corporate priorities may shift, drastically altering the system requirements.

The incremental approach is appropriate for a dynamic environment in which long-range objectives are established, but short-range conditions change. Long-range objectives are divided into subprojects. When one subproject is accomplished, the next subproject is re-evaluated.

Evolutionary project management is an interactive process involving design, development and implementation. While this method appears to permit increased creativity and innovation, this appearance may be deceptive. As the driving decisions for the development occur at the detailed analysis level, there is no longer overall control at the level of organizational requirements. When the focus is on immediate technological innovation, the nature and impact of the system as a whole entity, and as an integrated element within the organization, may be ignored. For people caught up in the excitment of evolutionary development, it is difficult to shift gears and view the development from an organizational perspective.

In evolutionary development, a rigorous process of regular reviews and assessments must be established to monitor the development, assess its impact on the organization and to determine whether to continue or change the current development course of activity.

D.A.1.8 An Incremental approach will be utilized. The first phase to be developed will be the automated carousel storage. Upon satisfactory operationalization of this phase, the AGV (Automated Guided Vehicle) system will be developed. When the AGV system is operational, an automated palletization system will be developed.

3.4 Phase Two: Viewing System Functionality

Once the organization's objectives have been established, and the current operations understood and documented, work begins to develop a system's functionality (see Figure 3.2 and Figure 3.3). Development of the functionality requires intense effort. Users, sponsors and producers contribute to this process. If the work is completed by members of a single group, crucial omissions or inaccurate assumptions weaken the soundness of the system.

3.4.1 Who Determines Functionality?

Sponsors and users are often excluded from this process because of timidity or the belief that non-technicians do not have the required skills. Some people may consider themselves, or their supervisees "too busy" to participate. The system suffers when users and sponsors do not participate in front-end planning.

3.4.2 What Are the Functional Parts of the System?

The Functional Description is a narrative about what enters the system, how it enters the system, what happens to it within the system and how it leaves the system during normal and abnormal conditions (see Table 3.2). Physical layouts are described and displayed with scaled layouts. Processes which require decisions may be described using flow diagrams which concisely express the functional options.

Table 3.2 Questions to Ask About a System's Functionality.

1.) What goes into the system?

2.) What does the system do with this input?

3.) What does the system use to accomplish its activity?

4.) What parts are released from the system?

3.4.3 Decimal Linkage Scheme

A decimal linkage scheme labels each component within the Functional Description. Each component is assigned a sequential, hierarchical label (see Figure 3.4). Each functional component's label is later assigned to the structured analysis blocks, the test plan scenarios, and tables and charts which may be produced to evaluate or improve functionality.

Cross-tables ensure that each functional component is correctly represented in all subsequent development activity. This linkage scheme reduces the number of structures which duplicate functionality, prevents the possibility of overlooking functionality, and ensures that all functionality is adequately tested and evaluated.

This decimal linkage scheme is particularly valuable during complex developments. Managing complexity is a task we are just starting to understand.

```
                 ___________   _____________   _____________
                |           | |             | |             |
                | A. PLAN   ----> B.PRODUCE ----->C. MAINTAIN |
                |___________| |_____________| |_____________|

  ______________   ______________   _______________   _______________
 |              | |              | |               | |               |
 |A.1 IDENTIFY ---> A.2 DEFINE   ---> A.3 STRUCTURED ---> A.4 TEST PLAN|
 |  OBJECTIVES  | | FUNCTIONALITY| |    DESIGN      | |   SCENARIOS   |
 |_________   __| |______________| |___________/\__| |_______________|
          ::                                   ::
          ::                             ......::...............
          ::                             : D.I.2.  Functional :
          ::                             : Analysis Documents :
          ::                             :...../\.............:
          ::                                   ::
      ___::________________                _____::_____________
     |   \/                |              |                    |
     | Users identify      |              | Users signoff      |
     | all functional      |              | Functional Analysis|
     | characteristics     |              | Documents, D.I.2., |
     | of a system which   |              | agreeing that they |
     | meets objectives    |              | will be able to use|
     | of organization.    |              | this functionality.|
     |                     |              |                    |
     | A.2.1 ANALYZE       |              | A.2.5 APPROVE      |
     | FUNCTIONALITY       |              |       ANALYSIS     |
     |___  ________________|              |_______________/\___|
          ::                                              ::
          ::                                              ::
 _________::__________   ____________________   __________::_________
|         \/          | |                    | |                     |
| Write Functional    | | Rigorous review    | | Brainstorm. Improve|
| Analysis Documents  | | and walk-through   | | the functionality. |
| * Process Analysis  | | of Functional      | | Challenge existing |
| * Data Descriptions| | Documents. Are     | | plans. Collaborate |
| * Logical Algorithm| | they complete,     | | to envision a more |
| * Software Specs    | | realistic, clear   | | useful & less      |
| * Hardware Specs    | | and unambiguous?   | | costly system.     |
|                     | |                    | |                    |
| A.2.2 DESCRIBE      | | A.2.3.TEST         | | A.2.4 IMPROVE      |
|       ANALYSIS   :::::>  ANALYSIS       :::::>    ANALYSIS        |
|_____________________| |____________________| |____________________|
```

A.2   DEFINE FUNCTIONALITY

Figure 3.2 Define Functionality A.2.

```
                   ___________    __________    ______________
                  | A. PLAN ----> B.PRODUCE ----> C. MAINTAIN  |

 ____________    ____________    _____________     ____________
|A.1 IDENTIFY ---> A.2 DEFINE  ---> A.3 STRUCTURED ---> A.4 TEST PLAN|
|  OBJECTIVES |  | FUNCTIONALITY|  |    DESIGN     |  |  SCENARIOS  |

 __________     __________     __________     __________     __________
| A.2.1     |  |A.2.2     |  |A.2.3      |  |A.2.4     |  |A.2.5      |
|DEFINE     --->DESCRIBE  --->TEST       --->IMPROVE   --->APPROVE    |
|FUNCTIONS  |  |FUNCTIONS |  |FUNCTIONS  |  |FUNCTIONS |  |FUNCTIONS  |
      ::                                 /\
      ::                                 ::....................
      ::                                 : D.I.2.  Functional :
      ::                                 : Analysis Documents :
      ::                                 :...../\.............:
      ::                                        ::
   ___::______________                        __::________________
  |   \/              |                      |                    |
  | Write Process     |                      | Users signoff      |
  | Analysis. Use text |                     | Functional Analysis|
  | to describe flow of|                     | Documents, D.I.2., |
  | all processes. Show|                     | agreeing that they |
  | relationships among|                     | will be able to use|
  | processes.  Label. |                     | this functionality.|
  |                   |                      |                    |
  | A.2.2.1 WRITE      |                     | A.2.2.5 APPROVE    |
  | PROCESS ANALYSIS   |                     |       ANALYSIS     |
  | DOCUMENTS          |                     |______________/\___|
       ::                                                  ::
       ::                                                  ::
 ______::________     ____________________     __________::________
|      \/        |   |                    |   |                    |
| Describe required | | Rigorous review   |   | Brainstorm. Improve|
| Data characteristic| | and walk-through |   | the functionality. |
| Data Dictionaries, | | of Functional    |   | Challenge existing |
| Record Descriptions| | Documents. Are   |   | plans. Collaborate |
| File size, Database| | they complete,   |   | to envision a more |
| Describe data with | | realistic, clear |   | useful & less      |
| charts/tables/text | | and unambiguous? |   | costly system.     |
|                  |   |                    |   |                    |
| A.2.2.2 WRITE DATA | | A.2.2.3. TEST    |   | A.2.2.4 IMPROVE    |
| ANALYSIS FOR DATA :::::>  ANALYSIS     :::::>    ANALYSIS        |
| DESCRIPTION & FLOW | |__________________|   |____________________|
```

A.2.2 DESCRIBE FUNCTIONALITY

Figure 3.3 Describe Functionality A.2.2.

I. Warehouse System Maintenance
Operators must be able to start up and shut down system.

I.1 System Start-Up
The material handling system will not be active at all times, but will alternate between being active and being shut down.

I.1.1 Types of System Start-Up
The start-up process needs to accommodate normal daily start-ups normal weekly start-ups and start-ups after abnormal shut downs.

I.1.2 Evaluate System Status Conditions
The start-up process should evaluate the status of all sub-systems, (Material Storage, Order Processing, Material Delivery).

I.1.3 Send start-up message to INCS (Inventory Control System)
When the start-up process is complete, an MSA (Material System Active) message is sent to the Inventory Control System.

I.2. De-activate Portions of System
The operators need to be able to de-activate any part of system.

I.3. System Shutdown
The operators need the capability to shut down the system. There are four types of normal shutdowns: end-of-day shutdown, end-of-week shutdown, end-of-year shutdown, and planned maintenance Shutdown. There are two types of abnormal shutdowns: controlled emergency shutdown and uncontrolled system abort shutdown.

II. Store Material
The Material Handling System controls automated storage.

II.1 Process Material Announcements
The system receives ANRs, (Advanced Notification of Receipt), messages from INCS (Inventory Control System) which list tote (containter) numbers, part numbers, and quantity of parts.

II.1.1 Evaluate ANR messages

II.1.2 Accept Valid ANRs

II.1.3 Reject Invalid ANRs

II.2 Receive Material

II.3 Find Storage Location

II.4 Putaway Material

II.5 Acknowledge Material Receipt

Figure 3.4 Sample of Linkage Scheme.

3.4.4 Technical Description of the System

The technical characteristics of the hardware and software required to support the functionality are described in four subsections:

(1) Functions
(2) Interfaces with other systems
(3) Physical configurations
(4) System operations.

A complex system has multiple subsystems that must be integrated with other systems. These subsystems are listed and defined separately and then integrated in a hierarchical organizational chart. The different forms of hardware required to support these systems are also documented in charts.

The interface between systems often starts as an apparently simple exchange of data. If the transferred data is in the form of messages, an initial table of messages, their meaning, instigating time and condition, and response is prepared. When systems are related in a strictly one-way exchange and receipt of data, the interface may be simple, particularly if the exchange of data is always in a batch mode, and never has real-time consequences.

When there is an interchange of data, systems both receive and send messages or data files. When the interchange occurs in real-time, the functioning of each of the integrated systems is constrained and dependent upon the time, meaning and consequences of the interchange.

Slight miscalculations or ambiguous interpretations of the timing or subtle differences in expectations or interpretations of the meaning and ramifications of messages may destroy the stability and integrity of both systems. With the increasing technological capacity for integrated systems, the problems associated with defining the requirement specifications of the interface interdepencies become increasingly complex. The interface between two systems may evolve into a third system, more complex than either of the original systems.

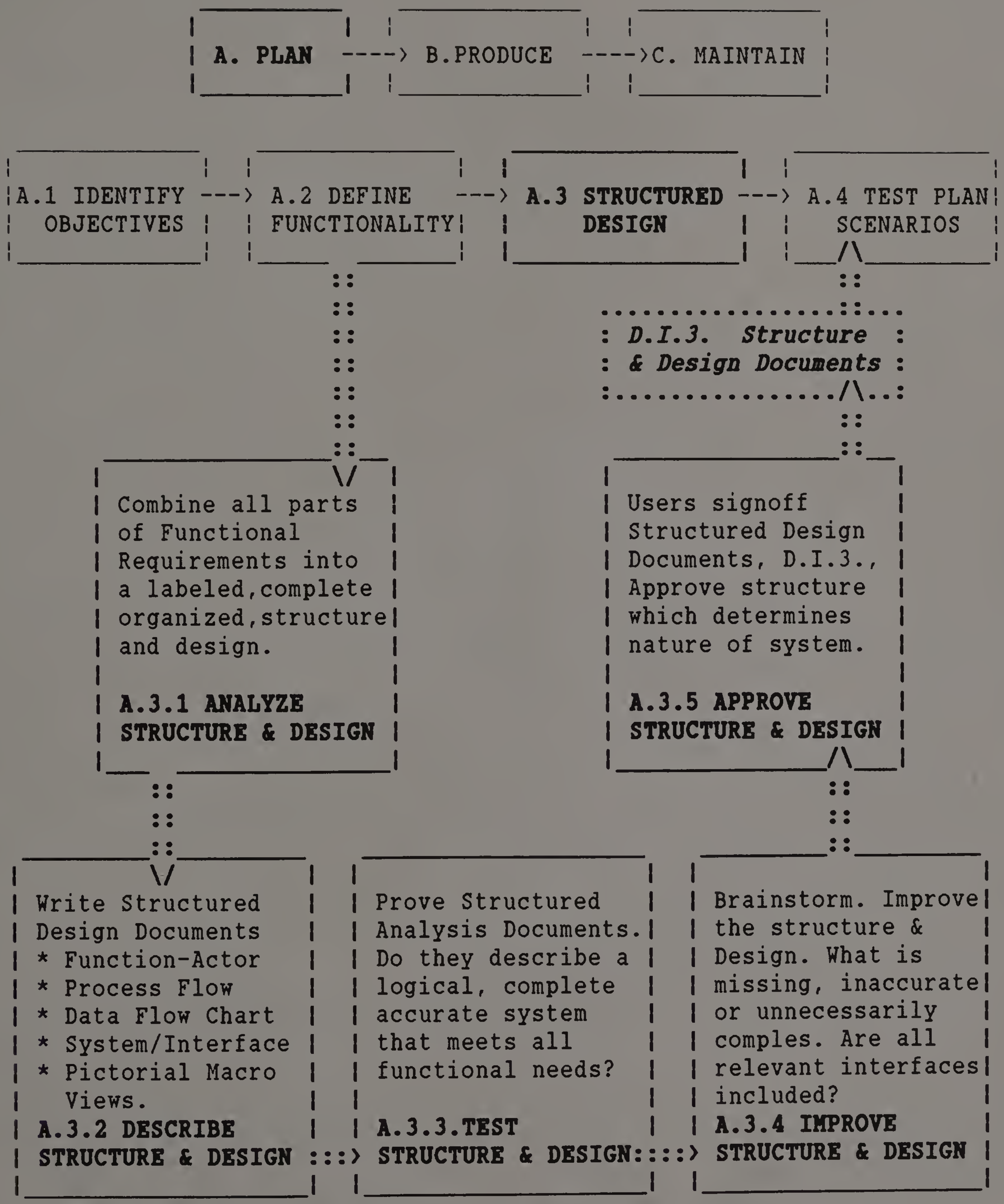
3.5 <u>Phase Three: Viewing the Structural Design</u>

In objective-directed design, the organization's objectives are the foundation of functional requirements. The analysis and design of the structure of the system supports and influences but does not determine the nature of the functionality. The operational implications of the proposed system's functionality derive from organizational objectives not from technological possibilities (see Figure 3.5 and Figure 3.6).

3.5.1 <u>Using the SADT Method</u>

There are three major stages in this phase. The first stage defines functional modules and data categories. The second phase structures the logical flow of system activity and of data. The third stage develops the requirements of programs which accomplish the activity and data flow processes. A powerful technique for structured analysis is the SADT^{tm} methodology, developed by Douglas Ross (1976).

With SADT, the system's functionality is categorized into three to six primary functional modules interconnected with directed paths of data flow, influencing factors and mechanisms.


A. PLAN
B.PRODUCE
C. MAINTAIN
A.1 IDENTIFY OBJECTIVES
A.2 DEFINE FUNCTIONALITY
A.3 STRUCTURED DESIGN
A.4 TEST PLAN SCENARIOS
D.I.3. Structure & Design Documents
Combine all parts of Functional Requirements into a labeled,complete organized,structure and design.
A.3.1 ANALYZE STRUCTURE & DESIGN
Users signoff Structured Design Documents, D.I.3., Approve structure which determines nature of system.
A.3.5 APPROVE STRUCTURE & DESIGN
Write Structured Design Documents
* Function-Actor
* Process Flow
* Data Flow Chart
* System/Interface
* Pictorial Macro Views.
A.3.2 DESCRIBE STRUCTURE & DESIGN
Prove Structured Analysis Documents. Do they describe a logical, complete accurate system that meets all functional needs?
A.3.3.TEST STRUCTURE & DESIGN
Brainstorm. Improve the structure & Design. What is missing, inaccurate or unnecessarily comples. Are all relevant interfaces included?
A.3.4 IMPROVE STRUCTURE & DESIGN

A.3 STRUCTURED ANALYSIS AND DESIGN

Figure 3.5 Structured Analysis and Design A.3.

```
                 ___________   ___________   _____________
                | A. PLAN -----> B.PRODUCE -----> C. MAINTAIN |
                 ‾‾‾‾‾‾‾‾‾‾‾   ‾‾‾‾‾‾‾‾‾‾‾   ‾‾‾‾‾‾‾‾‾‾‾‾‾

 ____________      ______________    ______________     ____________
|A.1 IDENTIFY ---> A.2 DEFINE    ---> A.3 STRUCTURED ---> A.4 TEST PLAN|
|  OBJECTIVES |   | FUNCTIONALITY|   |     DESIGN     |  |  SCENARIOS  |

 __________     __________     __________     __________     __________
| A.2.1     |  | A.2.2    |  |A.2.3       |  |A.2.4      |  |A.2.5       |
|ANALYZE    ---> DESCRIBE --->TEST        --->IMPROVE    --->APPROVE     |
|STRUCTURE  |  | STRUCTURE|  |STRUCTURE   |  |STRUCTURE  |  |STRUCTURE   |
       ::                         /\
       ::                    ...::..................
       ::                    : D.I.3.  Structure   :
       ::                    : & Design Documents  :
       ::                    :......../\..........:
       ::                              ::
   ____::______________             __::_______________
  |    \/               |          |                    |
  | Develop Table that  |          | Develop system     |
  | matches function    |          | macro view displays|
  | with action, with   |          | which offer simple,|
  | actor (person,      |          | pictorial views of |
  | machine,system)     |          | system's operations|
  | with data,programs  |          |                    |
  | A.3.2.1 WRITE       |          | A.3.2.5 DEVELOP    |
  | FUNCTION-ACTOR-DATA |          | PICTORIAL SYSTEM   |
  | PROGRAM TABLE.      |          | MACRO VIEW         |
  | DOCUMENTS           |          | DOCUMENTS.     /\  |
       ::                                          ::
       ::                                          ::
 ______::_________     ___________________    _____::_________
|      \/          |  |                    |  |                  |
| Develop structured | | Design structured  |  | Brainstorm. Improve|
| diagrams that    |  | data flow diagrams |  | the functionality. |
| describe all system| | to describe move-  |  | Challenge existing |
| functionality. Use | | ment, storage,     |  | plans. Collaborate |
| labeling technique.| | impact,transfor-   |  | to envision a more |
|                  |  | mation of all data |  | useful & less      |
|                  |  | processed by       |  | costly system.     |
|                  |  | system.            |  |                    |
| A.3.2.2 WRITE    |  | A.3.2.3. DEVELOP   |  | A.3.2.4 DEVELOP    |
| STRUCTURED PROCESS::::>  DATA FLOW      :::::> SYSTEM FLOW &      |
| FLOW DIAGRAMS    |  |    DIAGRAMS        |  |  INTERFACE CHART   |
```

A.3.2 DESCRIBE STRUCTURE AND DESIGN

Figure 3.6 Describe Structure and Design A.3.2.

The structured view fits the parts of the system (as defined in the functional analysis), together into a logically complete and structurally cohesive system. First, the functional modules are defined. Then a definition of the data which feeds these modules is developed to identify and define exactly what characteristics must be possessed by that data which enters the system (see Figure 3.7).

DATA NAME	*DATA TYPE*	*FORMAT*	*START POINT*	*END POINT*	*DEFINITION*	*REQUIRED?*
Box#	*Num*	*999999*	*010000*	*999999*	*Cart Number*	*Yes*
Part#	*Alph*	*XX-999*	*AA-001*	*ZZ-999*	*Part Number*	*Yes*
Cost	*Dec*	*ZZ9.99*	*000.00*	*999.99*	*Part Cost*	*No*
Numb	*Int*	*ZZZZZ9*	*000000*	*999999*	*Number of Parts*	*Yes*

Figure 3.7 Sample of Data Definition Table.

As the system's architecture develops, detailed inter-relationships and data definitions are expressed. When the structure is completed, the system is fully described in terms of its data, activity flow, data flow and hierarchical integration.

3.5.2 Matching Structured Design to the Functionality

Structured diagrams describe functionality using hierarchical blocks connected by processes represented by arrows. These blocks are labeled with a linkage system which clearly specifies each block's position in the overall scheme of the system. No functionality, originally included, is unintentionally modified or omitted in subsequent phases of systems development. By using the same numbering scheme in the functional documents, the structured blocks and the test plans, a one-to-one realationship between functional and structural components is rigorously maintained. Each design component is matched to a functional component.

3.5.3 Does the Structured Design Support the Functional Analysis?

Front-end planning evaluates the integration of functionality with structure. The developers design a macro structure of programs which captures the structured analysis envisioned in the SADT diagrams. Pseudo code defines functionality inherent in each individual program.

The data definitions, program specifications and design structures are compared with functional requirements to ensure compatibility. The process of fitting the structure to the conceptual is always led by functionality. The Function-Structure Table serves as a checklist for this fitting process (see Figure 3.8).

3.5.4 Trouble-Shooting with the Discrepancy Document

The Discrepancy Document is a tool for identifying, analyzing and resolving discrepant concepts. Ttrouble-shooting is a form of brainstorming which increases the system quality.

The development of a Discrepancy Document is a tedious yet critical process, requiring an observer who concentrates on innuendoes as well as overt statements (see Table 3.3). The process of discrepancy development involves a review of prior documents, notes and comments which occcured during the development. Scrutiny uncovers ambiguous interpretations of functionality and structure.

FUNCTION - STRUCTURE TABLE

Heading: Section I.2.1 *Topic: De-Activate Rotary Rack Shelf #1*

Actors	*Activity*	*Function*	*Module*	*Program*	*File*	*Fields*	*Message*
Op -> A	*Stop Rack1*	*I.2.1*	*A.2.1*	*P01051*	*Stat*	*Posit Loc*	
A -> TC							*HRN*
TC-> RR							*EN1*
RR-> TC	*Rack1 Off*						*OKEN1*
TC-> A	*Rack1 Off*			*P01052*	*Stat*	*Posit*	*OKR1H*
A -> Op	*Display Rack1 Off*			*P01053*			

Notes on Function - Structure Table:

Each section heading in the functional analysis is represented in this table. The "ACTOR" column identifies the actor(s) accomplishing each activity. Actors may be people, machines or system processes. For brevity, abbreviations are used to identify the actor: Op (Operator); A (System A); TC (Tracking Controller); RR (Rotary Rack)

If the action involves one actor, acting alone, the single initial appears under the heading of Actor. If the action involves interaction between actors, the initials are linked by arrows which indication the direction of the interaction:

Actor	Actor	Actor	Actor
Op (Operator Acting Alone)	Op->A (Operator Affecting System A)	TC<->RR (Tracking Controller sending messages to Rotary Rack & Rotary Rack sending signals to Tracking Controller)	A->Op A->TC (System A sending messages to both Operator & Tracking Controller.

The Activity Column describes the activity which is occurring. The Function Column lists the label of the section of the Functional Analysis which describes the functionality being accomplished. The Module, Program, File, Field columns link the activity to the specific elements of the system architecture which accomplishes the function. Message refers to the label of any message which is sent between programs or systems.

Figure 3.8 Sample of Function - Structure Table.

Table 3.3 Organization of the Discrepancy Document.

Section 1...	List of identified problems which need resolution.
Section 2...	Summarizes differences between concepts in functional document and interpretations of functionality.
Section 3...	Correlates text from conceptual functional diagrams with comments of development participants who identify discrepant interpretations.
Section 4...	Summary of decisions concerning discrepancies.

3.5.4.1 Identifying Ambiguous Interpretations

The essence of the specification process is human understanding, innovation, communication and judgment. The discrepancy process requires the additional ability to discern ambiguous interpretation of functionality. It offers a format for expressing functional differences which will be resolved during development (see Figure 3.9).

The person(s) responsible for identifying the discrepancies attempts to thoroughly understand the different interpretations, using an unbiased approach. Discrepancies exist--the challenge is to discover and resolve them before they disrupt the development.

Resolution of differences occurs by default if no attention is given to their resolution. Rhe resolution occurs by edict if a single person has unquestioned authority to resolve differences.

Programmers promote their own interpretations, often unaware of conflicts between their interpretations and that of users and of sponsors. When attempts are made to activate a system, discrepant interpretations surface at the most inconvenient and costly times.

Function.........	*II.1.2 Reject Invalid ANR (Advanced Notification of Receipt) messages.*
Problem..........	*System A's error handling differs from System B's expectation.*
Differences.....	*System A refuses to accept error messages.* *System B expects System A to accept error messages, placing them on hold until the problem is resolved.*
Functionality....	*II.1.2 Reject Invalid ANRs* *"Reject implies do not store in data base" (System A's programmer).* *"Reject implies store but do not process" (System B's programmer).*
History..........	*09/05/88 Programmers of System A developed details of ANR rejection message.* *09/10/88 Programmers of System A wrote programs which ignore invalid ANRs. After the Rejection message is sent, the initial ANR no longer exists.* *09/15/88 Designed System B to remedy problem and send remedy back to System A.*

Figure 3.9 Sample of Discrepancy Document.

3.5.4.2 Discrepancies Escalate Conflict

When discrepancies remain hidden or are ignored during the specification stage, the opportunity for constructive resolution is lost. When the discrepancies emerge at the end of development, the resolution is often destructive. The sponsors and users must either accept a system with faulty functionality (one which does not work acceptably within the operating environment), allocate additional resources to rebuild part of an existing system, or drop the system.

With complex systems development, the cost of discrepancy resolution is one of "pay a little now or pay a lot later."

3.5.4.3 Managing Discrepancy Resolution

The approach used to resolve discrepancies determines whether the development thrives or fractures into disarray. The challenge of project management is to allow discrepancy identification, analysis and resolution--and to allow reconciliation of diverse viewpoints.

The Discrepancy Document resolves discrepancies. Different people review the document and insert hand written comments. These comments become ongoing commentary in the document, with the commentor clearly identified. The discrepancy resolution process focuses attention on differences early in the development process and permits a resolution *based on analysis, not on political maneuvering.*

3.5.4.4 Overcoming Cultural Misunderstandings

This process is particularly useful when the relationship among participants becomes charged with emotion because of cultural misunderstanding. Often overlooked in systems literature is the intense nature of the relationships among systems development participants. The structure of front-end planning lets people work together, even when their ideas and interpretations conflict. Front-end planning focuses attention on concrete facts. People are no longer competing with one another, trying to promote their own viewpoint.

Instead, people focus on a clearly specified issue, with a variety of alternative interpretations. The task is to resolve a concrete issue, not to argue over misinterpretations.

Discrepancies indicate that people are thinking, incorporating stated functionality with their personal storehouse of knowledge. Without different viewpoints, and the resolution of these differences, there would be little need for people in the development process.

3.6 Phase Four: Viewing Operations with Early Test Plan Scenarios

The fourth phase of front-end planning is the development of early test plan scenarios which activate the system design.

3.6.1 Role of Traditional Test Plans

The purpose of testing a system is to determine if the system does what it is supposed to do. Traditionally, system testing determines whether program code satisfies program design.

3.6.2 Role of Early Test Plan Scenarios

Early test plan scenarios determine whether a system, based on the functionality and design, would be useful (see Figure 3.10).

```
TRADITIONAL APPROACH

 ___________________________
|                           |
| ROLE OF SPONSORS & USERS  |
|_:___:_______________:___:_|
| :   :               :   : |
| O   F               U   : |
| b   u               s   : |
| j   n               e   M |
| e   c                   o |
| c   t   D   C   T       d |
| t   i   e   o   e       i |
| i   o   s   d   s       f |
| v   n   i   e   t       y |
| e   a   g   :   :       : |
| s   l   n   :   :       : |
|     i   :   :   :       : |
|     t   :   :   :       : |
|     y   :   :   :       : |
|_________:___:___:_______:_|
|         :   :   :       : |
| ROLE OF SYSTEM DEVELOPERS |
|___________________________|
```

```
EARLY TEST PLAN SCENARIO APPROACH

 ___________________________________
|                                   |
|     ROLE OF SPONSORS & USERS      |
|_:___:_______:___:___________:___:_|
| :   :       :   :           :   : |
| O   F       :   :           U   : |
| b   u       :   :           s   : |
| j   n       A   I           e   M |
| e   c       c   m               o |
| c   t       t   p               d |
| t   i   D   i   r   C   T       i |
| i   o   e   v   o   o   e       f |
| v   n   s   a   v   d   s       y |
| e   a   i   t   e   e   t       : |
| s   l   g   e   :   :   :       : |
|     i   n   :   :   :   :       : |
|     t   :   :   :   :   :       : |
|     y   :   :   :   :   :       : |
|_________:___:___:___:___:_______:_|
|         :   :   :   :   :       : |
|     ROLE OF SYSTEM DEVELOPERS     |
|___________________________________|
```

Figure 3.10 Early Test Plan Scenarios Improve System Usefulness.

3.6.2.1 Early Test Plan Scenarios Match Design to Functionality

When early test plan scenarios are developed before coding, they match the design to functional requirements. These same test plans are used to develop procedure manuals and documentation. After system coding, the test plan scenarios become system acceptance tests.

Examining the "fit" between a developed system and the operational environment traditionally occurs during system start-up. The system is activated within the environment. If it works satisfactorily, the development is successful. If the system fails to work satisfactorily within the environment at start-up time, the results are painful and expensive: the environment must be changed, the system changed or the system removed from the environment.

Early test plan scenarios activate specifications, asking how well the specifications will work within the organization's environment. People are needed who understand the organization's conditions (both normal and exceptional) and its unique characteristics. The useability of a system is measured by how successfully the people, machines and systems within this operational environment work together.

3.6.2.2 Script Format: A Powerful Communication Tool

The script format used to activate requirements and design is a powerful communication tool. This script language of screenplays satisfies several requirements for specification evaluation (see Table 3.4).

Table 3.4 Conditions for Successful Evaluation of Specifications.

(1) The functionality represented within the Requirement Specifications must be presented in a format which can be understood by those people knowledgeable in the operating environment.

(2) The functionality must be represented using a format which enables people to merge an understanding of the specifications with the operating environment.

(3) The functionality must be presented in a way that inidividual patterns of activity as well as overall processes can be evaluated.

(4) Individuals must be able to use their unique areas of expertise and scrutinize those sections of the functionality which directly concern their responsibilities.

(5) Persons evaluating specifications must have time, opportunity, motivation and concern to be careful, thoughtful and thorough. in their work.

Using a script format satisfies all of these conditions. People understand information when it is presented within a familiar metaphor. Test plans are scripts set within the organization's actual environment. Using the actual environment as the presentation metaphor, people can use their expertise and understanding of the environment when they evaluate the proposed system.

3.6.2.3 "Moving Pictures" Increase Understanding

Reviewing the overall tests presents a "moving picture" for evaluation. To trace an her/his particular responsibility, the individual is able to follow the activities of a single "character" within the script, to assess whether the activities associated with that individual are valid, and as useful and easy as they might be.

The familiar setting of these "moving pictures," and the simple language used to describe actions increase understanding--particularly important when users and developers speak different languages or are of different cultures.

3.6.2.4 Early Test Plan Scenarios Increase Personal Responsibility

Those people who scrutinize the test plans are the same people who will run the tests after the programs have been developed, and who will eventually operate the system itself. By collaborating during this planning stage, people are able to contribute to the design of their future work, to improve the present plans by suggesting changes which will increase the system's usefulness (see Figure 3.11).

When people have to personally sign-off portions of a test plan, they become accountable, they assume personal responsibility. People who will run the tests and will operate the system must sign test plans indicating that if the system, when developed, acts according to these test plans, they will be able to successfully test the system and to productively operate the system.

3.6.2.5 Stricter Control of Functionality

Early test plan scenarios, developed to evaluate functionality and structure, provide stricter controls of functionality during program development. The *what* of a system should not be defined by the technological *how*. The *how* may influence the *what* by limiting or expanding the functional and operational possibilities, but should not, in seclusion or by default, determine the *what* of a system which has the potential to change the nature of the organization itself.

3.6.2.6 Controlling the Change Process

Early test plan scenarios are dynamic. If technical requirements must change the nature of the system, the test plans must be modified to reflect that change. A critical constraint of the change process is that each modification to the test plan must be clearly identified, reviewed and accepted by sponsors, users and developers who initially accepted the test plan (see Figure 3.12). Early test plan scenarios support changes, but force control onto the change process.

```
                 ___________    _____________    _____________
                |           |  |             |  |             |
                | A. PLAN   ----> B.PRODUCE   ---->C. MAINTAIN |
                |___________|  |_____________|  |_____________|

 ______________    _______________    _________________    __________
|              |  |               |  |                 |  |          |
|A.1 IDENTIFY ---> A.2 DEFINE    ---> A.3 STRUCTURED ---> A.4 TEST --> B
|  OBJECTIVES  |  | FUNCTIONALITY|  |     DESIGN       |  | SCENARIOS |
|______________|  |_______________|  |_ _______________|  |__________|
                                       ::                  ___________
                                       ::    ..............|           |
                  :::::::::::::::::::::::    :D.I.4. Early ::::> B.     |
                  ::                         :Test Plan    : |  PRODUCE |
                  ::                         :Scenarios...:  |__________|
                  ::                              /\
       ______________::____                 _____::____________
      |              \/    |               |                   |
      | Operationalize all |               | Users, Sponsors   |
      | parts of system's  |               | and Developers    |
      | functionality as   |               | formally sign-off |
      | it will work in the|               | test plan scenarios|
      | actual conditions  |               | which become devel-|
      | of organization.   |               | opment agreement. |
      |                    |               |                   |
      | A.4.1 ACTIVATE     |               | A.4.5 APPROVE EARLY|
      |     FUNCTIONALITY  |               | TEST PLAN SCENARIOS|
      |___ ________________|               |______________/\___|
          ::                                              ::
          ::                                              ::
 _________::_________    ____________________    _________::_________
|         \/         |  |                    |  |                    |
| Develop Early Test |  | Team walk-through  |  | Brainstorm. Uncover|
| Plan Scenarios.    |  | of test scenarios &|  | problems, blunders,|
| Divide test into   |  | Individual review  |  | errors. What can go|
| sequences with     |  | Are all activities |  | wrong? What could  |
| increasing         |  | useful, possible   |  | improve system,make|
| complexity. Use    |  | and measurable?    |  | it simpler, more   |
| script format.     |  |                    |  | flexible, more     |
|                    |  |                    |  | useful?            |
| A.4.2 DEVELOP EARLY|  | A.4.3. TEST AND    |  | A.4.4 IMPROVE EARLY|
| TEST PLAN SCENARIOS:::> EVALUATE SCENARIOS::::> TEST PLAN SCENARIOS|
|____________________|  |____________________|  |____________________|


    A.4   EARLY TEST PLAN SCENARIOS HELP IMPROVE SYSTEMS BEFORE CODING
```

Figure 3.11 Early Test Plan Scenarios Help Improve Systems Before Coding A.4.

```
                 ________   __________   ____________
                | A. PLAN ---> B.PRODUCE ---> C. MAINTAIN |
                 ‾‾‾‾‾‾‾‾   ‾‾‾‾‾‾‾‾‾‾   ‾‾‾‾‾‾‾‾‾‾‾‾
  ____________    _____________    _______________    ______________
 |A.1 IDENTIFY ---> A.2 DEFINE   ---> A.3 STRUCTURED ---> A.4 TEST PLAN|
 | OBJECTIVES |  | FUNCTIONALITY|  |     DESIGN    |  |  SCENARIOS   |

  ___________   __________   ____________   __________   __________
 | A.2.1      | |A.4.2     | |A.4.3        | |A.4.4     | |A.4.5     |
 | ACTIVATE  ---> DEVELOP  --->TEST        --->IMPROVE  --->APPROVE   |
 |FUNCTIONAL | |TEST PLANS| |TEST PLANS   | |TEST PLANS| |TEST PLANS|
        ::                             /\
        ::                             ::....................
        ::                             : D.I.4.  Early Test :
        ::                             :      Plan Scenarios:
        ::                             :...../\.............:
        ::                                   ::
    ____::_______________                ____::_______________
   |    \/               |              |                     |
   | Define sequential   |              | Develop test        |
   | segments for test.  |              | sections to stress  |
   | Divide test into    |              | system's capacity   |
   | acts for each cycle |              | to survive high     |
   | and any other       |              | volumes&exceptional |
   | critical situations |              | conditions.         |
   |                     |              | A.4.2.5 DEVELOP     |
   | A.4.2.1 DEFINE      |              | STRESS TESTS FOR    |
   | TEST SEGMENTS.      |              | SYSTEM              |
   |___   _______________|              |_CAPACITY.______/\___|
        ::                                              ::
        ::                                              ::
 _______::________     ____________________     ________::________
|       \/        |   |                    |   |                  |
| Assign all parts of| | Assign data values |   | Complete detailed |
| Functionality to   | | that represent     |   | scenarios of all  |
| one or more segment| | range & variety    |   | test functionality.|
| Include what should| | expected in actual |   | Who does what to  |
| labeling technique.| | operations.        |   | whom, where, when,|
| and what should not| | Calculate all data |   | for how long. Link|
| occur.             | | values system will |   | acts to functions |
|                    | | later calculate    |   | with labelis.     |
| A.4.2.2 ASSIGN     | | A.4.2.3. ASSIGN    |   | A.4.2.4 PRODUCE   |
| FUNCTIONALITY TO  :::::>  DATA AND       :::::> DETAILED TEST     |
|_TEST SECTIONS.____| |_CALCULATE CHANGES_|   |__PLAN SCENARIOS.__|
```

A.4.2 DEVELOP EARLY TEST PLAN SCENARIOS

Figure 3.12 Develop Early Test Plan Scenarios A.4.2.

3.6.2.7 Early Test Plan Scenarios Guide Coding

Many of the programmers who code a system do not participate in setting system objectives, the functional analysis or the system design. These programmers must rely on their personal interpretation of functional and design documents. It is the programmer who determines when she/he needs assistance in interpreting a condition. Test scenarios offer additional, explicit guidelines which reduce programmer misinterpretation of system requirements.

3.6.2.8 Early Test Plan Scenarios Become Acceptance Agreements

The early development of test plans establishes quantitative and explicit measurement criteria to evaluate the successful completion of the system's development. As seen above, these tests help guide the coding process to support the functional and design decisions.

Early test plan scenarios effectively guide coding because they are acceptance agreements--the formal acceptance tests of the entire development. *Early test plan scenarios are used by programmers to interpret specifications because the quality of their work will be directly measured by how well their programs satisfy these tests.*

3.6.3 Attributes of Early Test Plan Scenarios

The early test plan scenario's view incorporates operational considerations such as time, location, choice, variation, volume and exceptional conditions, into the system plans (see Table 3.5).

Table 3.5 Attributes of Test Plan Scenarios.

Temporal	Spatial	Environmental	Volume	Choice
Sequence	Location	Exceptions	Size	Flexibility
Time	Physical motion	Power Failures	Number	Errors
Timing	Change in location of parts.	Delays	Frequency	Changes

3.6.3.1 Early Test Plan Scenarios Based in Activity Cycles

The bases of early test plan scenarios are cycles of activities. Before developing test plans, the critical cycles within an operating environment must be identified. An initial plan is designed to test the activity which occurs during the shortest cycle. This test is expanded to incorporate all other relevant cycles (see Figure 3.13).

Cycles of a Manufacturing Corporation

```
1. Fiscal year
   2. Calendar months
       3. Work weeks
           4. Shifts
              5. Half-shift
                 6. Hours
```

Figure 3.13 Sequential Cycles of a Manufacturing Corporation.

3.6.3.2 Importance of Timeliness

Early test plan scenarios incorporate an operational environment's measure of timeliness importance. In an environment where the system has real-time functionality, the timeliness of the system's response is crucial. One hour of the test must duplicate the variety, volume, sequence and time requirements of interactive activities which will occur in a single, operational hour. If the environment's activity is structured into hours and shifts, the test may represent the activities of each hour in a single shift, the start-up and ending activities of a day, a week, a month and a year.

In a real-time environment, the system activity must successfully be started and completed, adhering to a particular sequence, within time boundaries of minutes and often seconds. In a batch mode environment, time boundaries may be extended to units of days, weeks, and sometimes months. Sequences of activities within batch modes remain critical--the system cannot print payroll checks until the hours worked have been entered into the system.

The test plan includes additional functional complexity in each iteration of a temporal cycle. The successful completion of one segment is assumed to occur prior to the start of all subsequent cycles. During the initial review of the tests, people track the system's operation and evaluate the organizational impact of these operations. When these test plans are used later in the development, they will include tools which permit the simulation of the successful accomplishment of prior steps.

3.6.3.3 Normal and Exceptional Conditions Tested

Depending upon the organization's operations, tests are developed to test the system's capacity to handle required volumes, to recover from external problems such as power loss, to adjust to changes in requirements, and to handle problems such as system crashes.

Operational test plan scenarios are designed to exercise the system's capacity to operate under normal environmental conditions. The tests are also designed to submit the system to abnormal, disruptive conditions. The inclusion of potential disruptions becomes a guide for the subsequent system development and for establishing system acceptance criteria. A system should be able to accomodate some, but not all, potential disasters. If power fails, the system may need to be able to switch into a manual mode.

3.6.3.4 Early Test Plan Scenarios Reduce Impact of Disagreements

The potential for major disagreements and conflicts between developers, users and sponsors is reduced when the system's expected response to problem situations is agreed upon prior to, and included within, the final system production contract.

3.6.3.5 Users Identify Unanticipated Problems

People who understand the operating environment contribute to the brain-storming required to identify potential exceptional conditions. There are often situations, beyond *expected* mistakes or fluctuations, which occur in an environment which outsiders do not anticipate. In a manufacturing system, there may be highly volatile material which cannot be placed onto a conveyor. A shipment of parts from a vendor may be labeled as containing fifty pieces per box, but when opened, after the box has entered the system, may have only forty pieces in each box.

If the unusual situation is commonplace (for example the wrong number of parts in a vendor's box), then the organization will have to address the problem prior to implementing the system. If the system recovery for this condition is slow in the proposed system, the development team will have to decide on alternate recovery schemes. The alternative might be to redesign the system to more quickly accommodate problems. Additional functionality, which may slow down the overall processing time, may be needed.

3.6.3.6 Early Test Plan Scenarios Linked to Functionality and Design

Early test plan scenarios are linked to the functional analysis documents through the rigorous inclusion of the same linkage labeling techniques which matched the system structural documents with the functional analysis documents. Each scenarios is also directly linked to the corresponding design component.

In an era of automated tools, this linkage can be simple to maintain. This linkage strengthens the traceability of system concepts--and the discipline of the development environment.

3.6.4 Testing the Usefulness of System Functionality and Design

Early test plan scenarios test a system's functionality and design during the most efficient phase--before coding. Blunders are not perpetuated and concretized in code. Mistakes and errors are corrected before they disrupt the development process and the system itself.

Early Test Plan Scenarios activate the system's functionality with the organization's people, machines and other systems. The developers of the test plan construct detailed scenarios of the system's actions and interactions with people, machines and systems as they occur in the organization's operational environment. During the Acceptance Test process, when tests are exercising the actual programs, reviewers ask how well the system's programs satisfy the test expectations.

During this earlier phase, of requirement specification testing, the reviewers' focus is essentially different. Reviewers focus on the useability of the functionality described by the test plans. Do the scenarios correctly fit the nature of the operational environment and satisfy the objectives of the organization?

Will a system which is able to satisfactorily pass all of the conditions expressed in this test, "work"? Will such a system improve and not harm the current organizational activities? Will the system be useful? Will the system be useable? Will the system be used? Specifications are challenged, and evaluated to ensure their adequacy for meeting the needs of the organization itself.

3.6.5 Developing Early Test Plan Scenarios

Early test plan scenarios are developed in the same, structured manner that the system's objectives, functionality and structure were developed. The components of the test plan are exactly linked to components of the prior planning documents. The front-end approach of planning, doing, evaluating, improving and approving is repeated during the development of early test plan scenarios.

Each of the following subsection titles will include the label associated with the task. The full diagram of this phase was previously described in Figure 3.12.

3.6.5.1 Define Test Sequential Segments A.4.2.1

The first step in writing early test plan scenarios is to define test wequential segments (or "acts") which match cycles within the operating environment. Segments represent cycles that characterize the operational environment. Additional segments are defined to represent significant events such as the end of the fiscal and/or calendar year, weekly start-up or shut-down and truncated weeks (see Figure 3.14).

SEGMENTS:	*Hours of Half-shifts.*
PURPOSE:	*The normal cycle for material storage, ordering and delivery are defined by hours. All orders must be closed within the 4 hour half-shift.*
SECTIONS:	*Hour 1..System Start-up; Initial load of material.*
	Hour 2..Complete ordering cycle.
	Hour 3..Complete ordering cycle.
	Hour 4..Complete ordering cycle; Close out all orders; System shutdown.
	Hour 5..Change storage location algorithm; Change delivery algorithm; End of year processing.

Figure 3.14 Sample of Test Segments for a Manufacturing System.

3.6.5.2 Assign Specific Functionality to Test Sections A.4.2.2

Assign each functional requirement (represented in the Functional Analysis Documents) to one or more segments. Include conditions for what should and should not occur. Begin simply, with single, normal functions. As the test progresses, overlap functions and introduce changes, error conditions and exceptions which represent expected situations within the actual operating environment.

3.6.5.2.1 Initial Outline of Functionality

Create an initial test outline which organizes functionality within the cycles of the test (see Figure 3.15). The first segment includes the basic, initial input into the system. Subsequent segments increase the number, combinations and complexity of activities occurring during a cycle. A final segment closes all activities. Other segments are developed to exercise occasional events such as fiscal year close-outs. Exceptional conditions are interspersed throughout the test. The first hour/segment is reserved for normal, unexceptional conditions. Error and problem conditions are incorporated into subsequent segments.

Hour	*Initial Repeat*	*Occasional Regular Exception Problem*	*Functional Activity*
1	*Initial*	*Occasional*	*Start-up System*
	Initial	*Regular*	*Receive valid Tote Announcement Message*
	Initial	*Regular*	*Receive valid Totes*
	Initial	*Regular*	*Putaway totes into automated storage*
	Initial	*Regular*	*Send Material Receipt Acknowledgement Message*

Figure 3.15 Manufacturing System's Initial Functional Outline.

3.6.5.2.2 Develop a Detailed Outline of Each Hour's Functional Activity

Using the initial functional outline as a guide, expand the original outline with more details on each functional requirement. Labels used to identify requirements in the functional analysis are included in this detailed outline (see Figure 3.16).

Matching Test Segment to Functional Label and Functional Description

Test Segment	*Function Label*	*Functional Description*
		Hour 1
H.1.0	*I.1.0*	*Send System Available Message.*
H.1.1.1	*I.1.1*	*Receive valid Advanced Notification of Receipt (ANR) messages.*
H.1.1.2	*I.2.1*	*Receive valid totes which were announced.*
H.1.2.1	*I.3.1*	*Find storage location for received totes.*
H.1.2.2	*I.4.1*	*Order totes putaway into storage location.*
H.1.2.3	*I.5.1*	*Acknowledge putaway completion.*
H.1.2.4	*I.6.1*	*Send Material Receipt Acknowledgement (MRA) messages.*
		Hour 2
H.2.1.1	*I.1.1*	*Receive valid and invalid ANR messages.*
H.2.1.2	*I.1.2*	*Reject ANRs with tote #'s outside of valid range.*
H.2.1.2.1	*I.2.1*	*Reject invalid 5" tote numbers.*
H.2.1.2.2	*I.2.2*	*Reject invalid 9" tote numbers.*
H.2.1.2.3	*I.2.3*	*Reject invalid 12" tote numbers*
H.2.1.3.1	*I.1.1*	*Receive valid totes which were announced.*
H.2.1.3.2	*I.2.2*	*Reject totes which were not announced by messages.*
H.2.2.1	*I.3.1*	*Find storage location for received totes.*
H.2.2.2	*I.4.1*	*Order putaway of received totes into storage location*
H.2.2.3.1	*I.5.1*	*Acknowledge putaway completion.*
H.2.2.3.2	*I.3.1*	*Locate new putaway location if first location is inaccessible.*
H.2.2.4.1	*I.6.1*	*Send MRA messages.*
H.2.2.4.2	*I.6.2*	*Send MRA for delayed location completions.*

(Note: There are three additional hours of detailed outline in the actual test).

Figure 3.16 Detailed Outline of Functional Activity.

3.6.5.2.3 Match Requirements, Structure and Test Plans

The purpose of early test plan scenarios is to improve a system's plans. It is critical that all of the functionality represented in the Functional Analysis Documents is directly represented in the test plans. After the detailed outline of the test plan is completed, a table is constructed which lists all sections of the Functional Analysis Documents and identifies the section(s) of the test plan where that functionality is specifically tested (see Figure 3.17).

Functional Description	*Functional Analysis*	*Structured Analysis*	*Operational Test Plans*
Receive valid ANR (Advanced Notification of Receipt) messages	*I.1.1*	*A.1.1.1*	*H.2.1.1.*
Reject ANRs with tote #'s outside of valid range:	*I.1.2*	*A.1.2*	*H.2.1.2.*
5" totes: 5000 - 8999	*I.1.2.1*	*A.1.2.1.*	*H.2.1.2.1.*
9" totes: 9000 - 11999	*I.1.2.2*	*A.1.2.2.*	*H.2.1.2.2.*
12" totes: 12000 - 12999	*I.1.2.3*	*A.1.2.3.*	*H.2.1.2.3.*

Figure 3.17 Matching Functionality, Structure and Test Plans.

3.6.5.3 Assign Data for Testing and Calculate Values A.4.2.3

Assign specific data values which represent the range and variety of values expected during operating conditions. All values that the system is expected to generate, are calculated by the test designers. The definition of these input materials should involve those users, sponsors and producers who know the possibilities, within the operating environment, for the normal range of values and for erroneous, redundant and exception data.

When constructing a table of input materials, include material which is characteristic of normal input into the system as well as material which exhibits all characteristics of exception input which should be considered (see Figure 3.18). Exception input includes data and material which the system should reject as well as that unusual input which it must accept. Test material is explicitly described. Because the test will be used to evaluate the system, all test material must be acquired and reserved for the acceptance test process.

ANRs (Advanced Notification of Receipt Messages) for Test Hour 1:

ANR#	TOTE#	INVENTORY#	QUANTITY	PALLET#
401	5401	AB-12345	50	400
402	5402	AB-12345	50	400
403	8403	AB-12345	50	400
404	5404	--------	50	400
405	5406	XY-23456	50	400
406	5435	AB-99999	50	400
407	5800	AB-12345	50	405
408	9001	DD-11111	5	1405
409	12001			

Totes for Test Hour 2:

ToteSize	Tote#	Contents	Pallet#	Comments/Condition	Accept?
5"	5001	60 - part AVC	100		Yes
5"	5002	59 - part AVC	100	1/2 label ripped off	Yes
5"	5002	60 - part AVC	100	Duplicate label	No
9"	5800	60 - part AXZ	100	Invalid tote# for 9"	No
9"	9001	25 - part X231	100		Yes
12"	12001	100- part S99	100		Yes
12"	12099	100- part RRE	---	No pallet	No

Figure 3.18 Data for Material Handling Test Plan.

3.6.6 Produce Explicit, Detailed Test Plan Screnarios A.4.2.4

Test plan scenarios describe *who/what* does *what* to *whom/what*, when, *where*, *how*, *why*, *when*, and for *how long*. All test plan activity is linked to functional requirements and structured analysis via labels and functionality cross-tables. The test plan includes the explicit ranges of acceptable values and boundaries for all scenarios.

The process of developing test scenarios activates the system's requirements and structure. It forces the developers to rigorously describe the step-by-step operationalization of each specification.

Each segment of the test plan begins with an overview of functionality in the segment (see Figure 3.19). By focusing attention on the new functionality, new elements are highlighted while repeat requirements are retested in more complex ways.

Test procedures associated with each section of the test segment describe prerequisites (see Figure 3.20), summaries of functionality (see Figure 3.21.), outline of functionality (see Figure 3.22), procedures (see Figure 3.23) and status of all relevant files after the completion of the test segment (see Figure 3.24).

The test plan author writes a scenario. The scenario must be *clear* enough for people to understand, and *detailed* enough to define exactly what actions should occur. The test plans for complex systems will be extensive. The initial early test plan scenarios required six months of preparation. Automated test plan generators and re-cycled scenarios can reduce this time by eighty percent. The six volumes of tests represented six hours of system operations. These tests became the final acceptance test for the entire development [Eastman, 1986].

HOUR 2 OF SYSTEM A'S PROCESSING

Hour 2 is divided into three sections to process the receipt of messages announcing material and orders, to pick the material and to receive the material.

The functionality introduced in Hour 2 is the receipt of messages requesting order deliveries, and system generated requests to pick the ordered material from storage.

2.1. - Process ANRs & MDRs

In section 2.1, ANR (Advanced Notification of Receipt) and MDR (Material Delivery Request) messages are received from System B and processed. Processing an MDR includes allocating material to the MDR and assigning the material to a Pseudo Pallet which will trigger the inclusion of the material in the pick process.

Some of the MDRs in Section 2.1 require material which will be stored in section 2.3. These will be delayed until the third test hour.

2.2. - Picking Material

In section 2.2, material is picked from System A storage in pseudo pallet sequence. The picked material is then palletized, acute picked or cycle counted. After picking, material destined for delivery outside of System A is palletized and delivered to its assigned Pick-up and Delivery Destination.

2.3. Receive Material into System A

In section 2.3, additional material is received into System A's storage from System B. Tests 2.1 and 2.3 occur simultaneously. System A's functionality requires the postponement of allocation as long as possible, yet still meet the two hour delivery requirement. Therefore, some of the material received during test 2.3 should be stored early enough for allocation during this hour.

Figure 3.19 Overview of Functionality in Test Plan Segment.

PREREQUISITES TO 2.2.1. - ACTIVATE PICKING PHASE

1. *Operators are at palletizing stations ready to begin palletizing totes as they arrive on conveyors.*

2. *Operators are at the flow rack palletizing stations ready to begin palletizing boxes as they arrive on the flow rack conveyors.*

3. *Operators are at the flow rack acute pick stations ready to begin processing acute pick and cycle count orders.*

4. *Operator is at the tote acute pick station ready to begin processing tote acute pick orders.*

5. *Operator is at the tote cycle count station ready to begin processing tote cycle count orders.*

6. *Two 5" totes, #13904 and #13905 are at the tote acute pick area.*

7. *Two containers, # 108536 and 108537 are taken to the flow rack acute pick area.*

8. *Status of Container File prior to test 2.2.2:*

Pallet#	*Tote#*	*HT-WD-LN*	*Part#*	*Lot#*	*Qty*	*Status*
PP-502	*11698*	*05-12-17*	*M892600*	*1101*	*10*	*ALLOC*
PP-502	*11701*	*05-12-17*	*M892600*	*1101*	*10*	*ALLOC*
PP-502	*11702*	*05-12-17*	*M892600*	*1101*	*10*	*ALLOC*
PP-502	*11877*	*05-12-17*	*M892600*	*1101*	*10*	*ALLOC*
------	*13904*	*05-12-17*	-------	----	*0*	*Empty*
------	*13905*	*05-12-17*	-------	----	*0*	*Empty*

(Note: Actual test contains several hundred entries in this file and several other files to explicitly describe all data in all files before the test begins).

Figure 3.20 Prerequisites for One Segment of Test Plan.

PREREQUISITES TO 2.2.1. - ACTIVATE PICKING PHASE

1. *Operators are at palletizing stations ready to begin palletizing totes as they arrive on conveyors.*

2. *Operators are at the flow rack palletizing stations ready to begin palletizing boxes as they arrive on the flow rack conveyors.*

3. *Operators are at the flow rack acute pick stations ready to begin processing acute pick and cycle count orders.*

4. *Operator is at the tote acute pick station ready to begin processing tote acute pick orders.*

5. *Operator is at the tote cycle count station ready to begin processing tote cycle count orders.*

6. *Two 5" totes, #13904 and #13905 are at the tote acute pick area.*

7. *Two containers, # 108536 and 108537 are taken to the flow rack acute pick area.*

8. *Status of Container File prior to test 2.2.2:*

Pallet#	*Tote#*	*HT-WD-LN*	*Part#*	*Lot#*	*Qty*	*Status*
PP-502	*11698*	*05-12-17*	*M892600*	*1101*	*10*	*ALLOC*
PP-502	*11701*	*05-12-17*	*M892600*	*1101*	*10*	*ALLOC*
PP-502	*11702*	*05-12-17*	*M892600*	*1101*	*10*	*ALLOC*
PP-502	*11877*	*05-12-17*	*M892600*	*1101*	*10*	*ALLOC*
------	*13904*	*05-12-17*	-------	----	*0*	*Empty*
------	*13905*	*05-12-17*	-------	----	*0*	*Empty*

(Note: Actual test contains several hundred entries in this file and several other files to explicitly describe all data in all files before the test begins).

Figure 3.20 Prerequisites for One Segment of Test Plans.

SUMMARY OUTLINE OF HOUR 2, SEGMENT 2 OF TEST PLAN - PICKING MATERIAL

Step 2.2.1 Activate Picking Phase

Material will be picked in pseudo pallet sequence. During the Site Acceptance Test, the picks for the rotary racks and the flow racks will be tested simultaneously.

Step 2.2.2 Acute Picks

A "Source" tote is the tote from which the material is picked during the acute pick process. The source totes for the acute picks should be the first totes picked from the rotary rack during a pick cycle.

Step 2.2.3.1 Picking of material from the rotary rack

No operator intervention is required during routine picking of material from the rotary rack. Operators must handle all totes which are sent to the rotary rack jackpot because of faulty processing.

Step 2.2.3.2 Picking of material from the flow rack

Pick lists will be printed to direct the manual picking of material from the flow rack.

Step 2.2.4.1 Manual palletization in the robotic palletizer area

At start up, there will not be robotic palletizers in use. However, the manual palletization process, which will be tested in the site acceptance tests, will duplicate the sequence and directions for the robopal. The system will control the exact sequence in which the totes are fed to the robotic palletization station.

Step 2.2.4.2 Manual palletization in the flow rack area

The palletization process in the flow rack area differs from the tote palletization. The system does not control the sequence of material to a palletization station.

Figure 3.21 Summary of Functionality in Test Plan.

OUTLINE OF FUNCTIONALITY OF 2.2.2 -- ACUTE PICK PROCESS PART 1/2

Acute Pick Process

1. *Allocation process has selected a source tote/container.*
   1. *Source tote/container has been inserted into a pseudo pallet record (record which contains pallet plan).*
   2. *Pseudo pallet record for source tote/container is the first pseudo pallet to be processed.*

2. *Source container is picked from rotary rack.*
   1. *Source tote delivered to rotary rack acute pick station.*
   2. *Source tote arrives prior to start of test.*

3. *Source container is picked from flow rack.*
   1. *Operator requests pick ticket.*
   2. *Pick ticket is printed which gives location of the source container in the flow rack.*
   3. *Operator removes source container from flow rack.*
      1. *Operator places source container on flow rack conveyor.*
      2. *Conveyor delivers source container to flow rack acute pick station.*

4. *Operator wands license number (bar code) of source tote/container (with optical scanner).*
   1. *System tells operator that the tote/container is for an acute pick.*
   2. *System asks operator to wand empty tote/container.*
   3. *Operator wands empty tote/container.*

5. *System tells operator how much to put into empty tote/container.*
   1. *Operator confirms actual quantity picked.*
   2. *Operator can adjust acute pick count.*
   3. *The empty tote/container which received material from the source tote/container becomes the acute pick tote/container.*

6. *System directs putaway of acute pick tote.*
   1. *System locates new rack location for acute pick tote.*
   2. *System tells operator to put acute pick tote on conveyor.*
   3. *System stores tote.*

7. *Acute pick container remains in flow rack area.*

Figure 3.22 Outline of Functionality in Test Plan.

PROCEDURE FOR 2.2.2 -- ACUTE PICK PROCESS   PART 1/3

Action	*Expected Response*	*Actual Response*
1. Operator awaits source tote in rotary rack acute pick area.	*1. System delivers tote# 11877 to acute pick area.*	*1.*
2. Operator wands source tote# 11877.	*2. System indicates acute pick and tells operator to wand empty tote.*	*2.*
3. Operator wands empty tote# 13904.	*3. System tells operator to place 3 pieces into tote# 13904.*	*3.*
4. Operator confirms the placement of 3 pieces in tote# 13904.	*4. System locates rotary rack storage for tote# 13904 and tells the operator to place tote# 13904 on rack's conveyor.*	*4.*
5. Operator re-wands source tote license.	*5. System indicates that another pick remains and tells operator to wand another tote.*	*5.*
6. Operator wands tote# 13905.	*6. System tells operator to put 5 pieces into tote# 13905.*	*6.*
7. Operator confirms the placement of 5 pieces into tote# 13905.	*7. System locates rack storage for 13905 and tells operator to place tote# 13905 onto rack's conveyor.*	*7.*

Figure 3.23 Procedure Page in Test Plans.

RESPONSE 2.2.2 AFTER HOUR 2 OF ACUTE PICK PROCESS PART 1/8

Status of Container File after test 2.2.2:

Pallet#	Tote#	HT-WD-LN	Part#	Lot#	Qty	Status
PP-502	11698	05-12-17	M892600	1101	10	ALLOC
PP-502	11701	05-12-17	M892600	1101	10	ALLOC
PP-502	11702	05-12-17	M892600	1101	10	ALLOC
PP-502	11877	05-12-17	M892600	1101	2	Avail
------	13904	05-12-17	-------	----	3	Alloc
------	13905	05-12-17	-------	----	5	Alloc

(Note, there are hundreds of entries in this file, in actual test).

Figure 3.24 File Description After Completion of Test.

3.6.7 Develop Stress Test for Operational Capacities A.4.2.5

Each system is vulnerable to certain types of stress. Material handling systems are vulnerable to large volumes of material which need to be processed within a short period of time. Systems which interface with other systems are vulnerable to breakdowns of the other systems. Systems which interact with machinery are dependent upon the machinery acting within certain boundaries. An accounting system may be particularly vulnerable to process interruption such as power cut-off during a general ledger update. Brainstorming develops "worst-case" scenarios for the proposed system.

Special tests may need to be constructed to test conditions which are expected to stress the system. These tests are run after the basic functionality has been proved. For example, a time-volume test for a material handling system will test the ability of the system to process high volumes of material within specified time boundaries.

The design of the time-volume tests follows the same format as the basic functionality tests, often using the same data and material. The difference between the two versions is that the basic functionality test has limited input in order to scrutinize specific parts of functionality while the stress test is scrutinizing the ability of the system to maintain functional viability while being stressed with high volumes or extreme time pressures.

3.6.8 Troubleshoot Activation of Functionality

During the development of early test plans scenarios, the functionality is scrutinized by the test plan designers. They "activate" each requirement, describing each functional element in terms of its operations.

The crucial stage of front-end planning occurs once the early test plan scenarios are written. Representatives of all groups who will be affected by the system troubleshoot the system's functionality, as it is activated by the scenarios.

The composition of the brainstorming team is critical. All people who have contributed to earlier viewpoints on system requirements are needed to ensure that the system supports all objectives, functional needs and structured analyses. This team includes representatives from each job function which will be needed to operate the actual system. Representatives must be knowledgeable about the operations. They should be the same people who will perform the acceptance tests after the system has been developed. Representatives should be the same people who will operate the system.

Each member of the troubleshooting team asks questions, trying to find problems with the described functionality (see Table 3.6). As each member of the team evaluates the overall acceptability of the proposed system, members are also assigned specific sections to examine extremely carefully.

Table 3.6 Evaluating Early Test Plan Scenarios.

(1) Do the test scenarios satisfy prior specification documents?

(2) Are all activities in the test plan scenarios possible?

(3) Are all activities in the test plan scenarios useful?

(4) Are all activities in the test plan scenarios measureable?

(5) What can go wrong?

(6) What problems will occur when people use the system?

(7) What risks will increase with this system?

(8) How is the system vulnerable?

(9) Would this system be used?

Those people who will (eventually) run the acceptance tests and operate the activated system are responsible for the useability of all sections of the test which will affect their work. Each individual team member signs off one or more segments of the test plan. Signing off confirms agreement with the operationalization of the future system, as it is represented in the accepted version of the test plan.

3.6.9 Improve Early Test Plan Scenarios

Troubleshooting early test plan scenarios rigorously exercise the simulated system operations to uncover problems, flaws and weaknesses. The next step addresses problems identified during the troubleshooting, and improves functionality and design.

The management of this stage requires the careful nuturing of an atmosphere of diligent team search for improvements. Individuals will disagree about potential improvements. An improvement according to one viewpoint will create new problems for another viewpoint. Individuals who designed parts of the specifications may not want to have their creations changed or eliminated.

The task of improving requirement specifications may be the most difficult aspect of system development. People evaluate specifications as they are represented, not as they are be personally envisioned. People have to challenge concepts, challenge scenarios. The concept of "fourth dimensional thinking" applies. Reviewers envision the system, as described in the early test plan scenarios, in the future environment of the organization. This vision merges the formally written scenarios with all of the information an individual has accumulated about the operational environment.

Skepticism, creativity and innovation are needed to envision how the system could operate more successfully than currently planned, and to translate these improvements into reasonable modifications.

Earlier specifications may be modified to reflect improvements. Extensive modifications will require substantial time. However, the time, cost and disruption required for modification of the system's design during this early stage will be substantially lower than if the same problems were identified after the system became operational.

3.6.10 Formal Signoff of Early Test Plan Scenarios

The signoff of the operational test plan scenario documents establishes a contract between sponsors, users and producers. The sponsors agree that the functionality represented in the test plans supports their organizational objectives. The users agree that the functionality will be useable during operations.

The developers agree that they will be able to develop a system, using the design established in the structured analysis documents, that will satisfactorily perform all of the actions and interactions described in the early test plans scenarios.

CHAPTER 4

FRONT-END PLANNING WITH EARLY TEST PLAN SCENARIOS IMPROVES COMPLEX SYSTEMS DEVELOPMENT

The real nature of things, that we shall never know, never. Einstein

4.1 <u>The Enigma of Systems Development</u>

Systems development remains an enigma. Most systems developments are failures--yet we still invest billions of dollars annually into these developments. System theories are rigid structures--yet programmers are notoriously unstructured. System researchers argue that poor specifications cause development failures--yet most research examines post-specification phases of coding and implementation. Systems are created by, for and of the people--and yet most development approaches concentrate on the technology.

4.2 <u>The Nature of Complex Systems</u>

The complexity of a computer system is measured in different ways. A complex system may be a "real time" system which responds immediately to a request from a person or other system. A complex system may be measured by the number of lines of code of its programs - when a program has more than a thousand lines of code, the program is "complex". A complex system may be interacting sets of programs.

4.2.1 A Definition of "Complex System"

The definition of "complex system" used in this thesis is: A set of programs, implemented on one or more computers, which affects the interaction of people, machines and other systems in an organization, and which is developed by one or more groups of persons who are not the actual users of the system.

4.2.2 Complex Systems Often Disrupt Operations

The injection of complex systems often disrupts the operations of an organization. "It is very difficult for automation to compete against a well-managed, highly-motivated work force" [White, 1987, p. 25]. Some complex systems help organizations, others disrupt organizations.

4.2.2.1 Invisible Impact

A complex system has an invisible impact [Moor, 1985, p. 273]. The invisibility of a system's data manipulation frees people from the necessity of constant attentiveness to overwhelming, redundant detail. Invisibility also leaves people vulnerable. People depend upon the results of unseen data manipulation without knowing the exact nature of the methods used to obtain these results, and with few tools to test the viability of these results. In 1988, the American presidential election was dependent upon complex vote tabulation systems which are not available for public scrutiny. The system's logical structure, the source code of all the directions for vote calculations, is owned by a private corporation, with no current opportunity for public scrutiny.

4.2.2.2 <u>Malleable Logic</u>

Complex systems have malleable logic [Moor, 1985, p. 269]. Data can be accepted and rejected, manipulated and presented in any way which is logically correct, with logic being defined strictly by a set of rules, devoid of meaning. A logically "correct" vote tabulation program could add one extra vote to a candidate's total whenever the total for the opposing candidate was evenly divisible by 63.

4.2.2.3 <u>Hominization</u>

A challenge facing developers is to specify systems which use the logical power of computers to help people more fully exploit human judgment, intuition and creativity. Hominization, (treating a machine as if it were human), occurs when users feel a system is displaying independent, human, or super-human, thought [Weizenbaum, 1976].

If users of complex systems do not understand the internal logic of a system and cannot predict the actions of a system, the mechanical, predetermined nature of computer logic seems overwhleming and appears to indicate a profound, independent nature. *If users attribute thought to a system, the system replaces the person as the decision-maker.*

This hominization of systems presents a dilemma to users. How can the users determine when to rely on a system's output, and when to challenge a system's omnipotence by using human judgment? The United

State's downing of an Iranian airline and the launching of the space shuttle during poor weather conditions [Stapleton and Putre, 1988] are tragic examples of the dilemma of needing to both trust and challenge complex systems.

> If we fail to put logic machines in their proper place, as aids to human beings with expert intuition, then we shall end up servants supplying data to our competent machines. Should calculative rationality triumph, no one will notice that something is missing, but now, while we still know what expert judgment is, let us use that expert judgment to preserve it. [Dreyfus and Dreyfus, 1986, p. 206].

4.3 <u>The Nature of System Maintenance</u>

A major portion of the time and cost associated with complex systems development occurs after a system is operational. System maintenance refers to those changes which fix or improve the useability of software which is in operation.

Software is extremely tractable. Simple changes often have unpredictable results which disrupt an entire system [Bryan and Siegel, 1984, p. 59]. A single, minor change may have the effect of a tiny hole in a dike which, once opened, destroys the integrity of the dike.

4.3.1 <u>Software Maintenance Corrects Design Blunders</u>

Software maintenance corrects design blunders--it is different from maintenance of most other types of products. Maintenance of non-software products includes replacing malfunctioning or obsolete parts. When system software is "maintained, its structure is changed. This "patch-work evolution" deteriorates the structural integrity and quality of the software system [Grady, 1987, p. 35].

4.3.1.1 <u>"Bugs" in Logic</u>

Complex programs contain "bugs" or errors in the logic of program code. Systems maintenance corrects logical errors. Logical errors reflect technical problems which may be discovered and corrected by skilled technicians.

While some logical errors are noticed early in a system's operations, others may not be revealed until the system has been running for months, or even years. A payroll system, running successfully for years, fails when an unexpected combination of data values occurs. Users are unable to run the payroll until the program is "maintained."

4.3.1.2 <u>Errors in Meaning</u>

When developers have incorrect interpretations of users' needs, they write programs which contain errors in meaning. "Productivity" in a health agency's management reports may mean "Staff Hours Worked" divided by "Staff Available Hours". The agency's director considers "Staff Available Hours" to be FTE (Full Time Equivalency) multiplied by the number of work days in a particular month, *including the number of leave hours taken by the staff person during the month.* The programmer assumes that a staff person's available hours *excludes any authorized leave time taken.*

The programmer's assumption *increases* the productivity of every staff person who uses authorized leave time. When errors in meaning are discovered, the user must accept the system's meaning, or pay the programmer to "maintain" the system.

4.3.2 Maintenance Improves Usefulness

The usefulness of a system may be improved in several ways. An existing function may be modified to make it more useful. The director of an agency which has several clinics may find that productivity reports, produced separately for each clinic, are useful only if they are consolidated. If manual consolidation is inefficient, the system is more useful after "maintenance" provides consolidated reports.

4.3.2.1 Functionality Added

Systems may require additional functionality which was unrecognized during the period of system specification. An organization may need to allocate expenses to a variety of cost centers based on fluctuating percentages. At the time of initial specification, the need for percentages was identified, but no member of the specification team recognized the organization's need to change these percentages. The need to permit fluctuation became obvious once the system became operational. Users were unable to adjust percentages. Because the initial system was based on non-fluctuating percentages, expensive modifications were required to "maintain" the system [Eastman, 1987].

4.3.2.2 Functionality Deleted

Systems may include too much functionality. Designers often resemble Mark Twain's man with a new hammer -- everything resembles a nail which should be hit. Systems with superflous "bells and whistles" functionality, no matter how technically intriguing and impressive, run slower and are more cumbersome to operate than simple systems which include only required functionality. Accounting systems which create useless reports waste time and resources. "Maintenance" is needed to reduce this system-imposed waste of resources.

4.3.2.3 Complexity Simplified

Communication between two systems may be too complex, requiring more counter-checks than are necessary for system validity. Superfluous error checks may force the selection of inappropriate options, disrupting an organization's operations. "Safeguards" to protect the system from human mistakes, may prevent people from acting responsibly in situations which need human judgment.

4.3.2.4 Unanticipated Changes

Changes in requirements may be anticipated or unexpected. Changes may be known but overlooked during specification. Unexpected changes destroy a rigid system. The market demand for a product changes. Technological advances make current operations obsolete. A parent organization alters the mission of a divisional plant. Unless a system's design is flexible enough to accommodate these changes, a costly rebuilding or total scrapping of the system may be the only options available.

4.3.3 Is It Maintenance or Is It Compulsion?

Some developers of complex systems believe that system start-up begins the major phase of development. "Tweaking" (adjusting) a working system is more exciting than the hard work of planning.

Programming offers immediate gratification. Programming can become a compulsive activity. An intriguing addiction afflicts many of us who become involved with computers. There are few compulsive planners, but there are a great many compulsive programmers [Weizenbaum, 1976].

4.3.3.1 Reactive Maintenance

Traditionally, systems developers assume a reactive approach to software maintenance, developing and using a system before attempting to increase the useability and effectiveness of the system [Grady, 1987, p. 35]. Reactive maintenance tends to reduce the quality of the system [Grady, 1987, p. 35]. Proactive maintenance, prior to system coding, can improve the quality of a system by decreasing complexity and increasing cohesiveness.

4.3.3.2 Perfection Unrealistic

Perfection is an unrealistic goal for developers of complex systems. The cost of building a "perfect" systems would be prohibitive [Bersoff, 1984, p. 79]. The goal is to build systems which are "good enough" to use. The realistic task of systems planning is to minimize, not eliminate, systems maintenance.

4.3.4 Reducing Future Maintenance with Test Plan Scenarios

Maintenance is the most expensive phase of systems development. Test plan scenarios reduce three major components of the maintenance phase by: (1) Correcting errors both in structure and in concept; (2) Customizing the system to be useful within the operating environment of the organization; (3) Improving the system through adding, deleting and modifying system functionality.

4.3.4.1 Reducing Errors in System Structure and Concepts

Errors in system concepts occur when developers are unable to build a useable system because they have an insufficient understanding of the functionality required by users. Reducing errors in system concepts receives little attention in systems research. Researchers appear to assume that erroneous concepts become visible as the users and sponsors review the developers' system descriptions. With complex systems, the system's description can fill several volumes. Structure and flow appear logically correct with an overwhelming amount of paths, patterns and detail. It is difficult to assess the system as a whole. We cannot evaluate the forest, only trees.

4.3.4.2 <u>Rigor of Early Test Plan Scenarios Identifies Errors</u>

The rigor of the test plan development and review identifies basic errors in concept and structure. Early test plans force participants to question the operational correctness of the system's specified structure and design.

4.3.4.3 <u>New Approaches Required to Reduce Errors</u>

Most systems development theories--and the associated research--focus on the logical correctness of the system and its development methodology.

The integrated systems of the 1990's force the focus onto operations. The increasing cultural diversity of the 1990's require that system development approaches acknowledge the individuality of participants. The overwhelming degree of system failure mandates that we learn how to improve the interaction of people during the development of complex systems.

4.4 <u>Theories of Systems Development</u>

There are many theories of system development. Formal, structured theorists argue that the purpose of systems development research is to discover, establish, and enforce rules and standards which could make the process of systems development free of human error [Mathiassen and Munk-Madsen, 1985]. Less traditional theorists consider systems development a craft which requires technical expertise, creative innovation and user-developer interaction [Macro and Buxton, 1987]. Researchers who focus on the interactive, social nature of systems development [Scacchi, 1984] argue that it is fundamentally a political process [Floyd, 1985].

4.4.1 Folkwisdom of Hints not Rules

Developers during the 1970's and 1980's favor hints of folkwisdom over rules of formal methodology. Unfortunately, these hints are too vague to manage the complex developments of the 1990's.

4.4.1.1 Approaches that Work--Sometimes

The enigmatic nature of software development forces researchers to acknowledge that their findings are based on their own experiences and are only guidelines, "approaches that work, sometimes" [Daly, 1977, p. 242]. Researchers offer principles to improve systems development. Gilb [1988] based his principles on insights he developed during a multi-million dollar system development failure by VOLVO of Sweden.

> * *The invisible target principle.* All critical system attributes *must* be specified clearly. Invisible targets are hard to hit (except by chance).
> * *The Clear-the-Fog-from-the-Target Principle.* All critical attributes *can* be specified in measurable, testable terms, and the worst-acceptable level *can* be identified.
> * *The Fail-Safe Minimization Principle.* If you don't know what you're doing, don't do it on a large scale [Gilb, 1988, p. 163].

4.4.1.2 Avoid Choosing a Terrible Way

Others who are concerned with improving the systems development process reject claims of "best" methods and try to help developers "avoid choosing a terrible way" [Lampson, 1984, p. 11]. Observations offer insights to help a developer chart his/her own course. These "hints" form a growing body of "folk wisdom" [Lampson, 1984, p. 11].

> * Too many automated systems are designed "for show not dough".
>
> * Systems are often designed on the basis of state-of-the-art solutions that exist currently or...in anticipation of advances in the state of the art. They are "solution driven" systems instead of "requirements driven" systems [White, 1987, p. 25].

4.4.2 Systems Development Life Cycle Theory

Software engineering produced the Life Cycle Theory, the first, most widely supported, theory of systems development. The Life Cycle theory assumes that the development of software programs can be understood and improved by application of scientific methodology.

> *Software Engineering*: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them [Boehm, 1976, p. 1226].

Software engineering tries to develop systems which are reliable, understandable, efficient and modifiable, using technical tools and principles such as modularity and confirmability [Ross, Goodenough and Irvine, 1975, p. 17]. The Life Cycle theory assumes that quality systems are developed by completing a sequential series of tasks (see

Table 4.1). The economic motive for the Life Cycle process is that the cost of software changes increases substantially as the development progresses from plan to code to operations [Boehm, 1984a, p. 4].

Table 4.1 Phases of Life Cycle Developments.

5 Phase Plan	7 Phase Plan
1) Planning	1) Feasibility
2) Specification	2) Plans and Requirements
3) Design	3) Product Design
4) Coding	4) Programming
5) Testing	5) Integration and Test
	6) Maintenance
	7) Phaseout
[Daly, 1976, p. 237]	[Boehm, 1984a, p. 4]

Most Life Cycle theories of systems development do not view change as an integral part of systems development. Those that do acknowledge the need for improvements, place maintenance after operationalization (see Figure 4.1).

```
                                                      Customer
                                                    Demonstration
                 Design                Develop-           /\
Customer         Review    Program     ment               ::    Operational
Requirements     Cycle     Coding      Baseline   Testing ::    Baseline
1 -----------> 2 ------> 3 ------> 3/4 -----> 5 -----> 6 --> 7
/\               /\                    /\         /\                ::
::<::Verification&Validation           ::         ::                ::
:: <::::::::::::::::::Change  Requests            ::                ::
                                       /\         /\                ::
                                Maintenance       ::                ::
                                       4 -------->Emergency         ::
                                       /\         Patches     Operational
                                       ::                      Incidents
                                       :: <:::::::::::::::: <:::::::
      CTEC's Software Development and Change Control Process
      [Bryan and Siegel, 1984, p. 63]
```

Figure 4.1 Role of Maintenance in the Life Cycle Process.

4.4.3 Multiview Specification Methodologies

The front-end planning approach described in this thesis suggests that the process of developing integrated systems which merge information processing with operations is too complex to rely on either folkwisdom or single-approach methods. Integrated systems development is improved with a multi-view specification process.

4.4.3.1 DSK and SEK Models

Arango and Freeman suggest two models for system specification. The DSK, Domain-Specific Knowledge model, represents the knowledge of those users and sponsors who understand the problem which must be solved. The SEK, Software Engineering Knowledge model, represents the software solutions for problems. Arango and Freeman argue that much of the difficulty associated with building useable systems is caused by incomplete DSK and SEK models, and the lack of integration between the two models [Arango and Freeman, 1985, p. 63]. This thesis uses early test plan scenarios to integrate the DSK and SEK models.

4.4.3.2 PQRST Model

Botting [1985] views software development as a multi-faceted undertaking which substantiates the existence and importance of different viewpoints during the systems development process. Botting assumes that an appreciation of different perspectives strengthens the development process by ensuring a more complete analysis, specification and design of the system prior to the actual construction of the system. Botting's eclectic approach, **PQRST**, requires different specifications for Purpose, Quality, Reality (static and dynamic factors of the environment), System (processes which will be improved by software) and Techniques/Technology.

This thesis improves Botting's eclectic methodology by integrating the five different specifications, verifying that they are compatible and mutually support a system which is useable within its operational environment.

4.4.4 Interactive Design Teams

Programmers alone should not make business decisions and users alone are unable to transform their needs into systems. Interactive design teams of users and programmers can be the catalyst for improving the planning and design phases of systems development [Leventhal, 1986, p. 49]. These teams emphasize collaboration among people with different skills, priorities and responsibilities.

A factor in systems failures, when users are isolated from the development process, is the perception of developers and sponsors that users are "individuals who need order and guidance in their lives and who are motivated by financial rewards rather than opportunities for personal growth" [Dagwell and Weber, 1983, p. 987].

Viable user participation requires users who are well-trained. "Participation is little more than hostage-taking, if the users are not given a training over and above the technical training traditionally offered by software vendors" [Bjorn-Andersen, 1985, p. 845].

4.5 Specifications Underpin Systems Development

Specifications underpin systems development. When specifications are incomplete, ambiguous or inaccurate, programmers build the system according to technical priorities. Technically "sound" systems are often unuseable [Potosnak, September 1987, p. 87].

4.5.1 Specification Blunders Are the Most Expensive Errors

Specification blunders are the most expensive errors made during systems development. Delays in identifying and correcting these errors substantially increase the cost of the development process. Escalated cost results from the difficulty of: (1) Uncovering errors, (2) Defining and analyzing problems, (3) Designing solutions which will not disrupt the rest of the system, (4) Coding to correct the error, (5) Adjusting the system to accommodate the modified code, (6) Testing the changes, and (7) Documenting the corrections [Lipow, 1979].

4.5.2 Who Specifies?

Researchers rarely ask "who specifies?" Few people question how a specification develops [Fickas and Nagarajan, 1988, p. 37]. Users are characterized as not "knowing what they need." Sponsors prefer to let the often highly paid developers "do the work." Sponsors may not want employees to leave their regular work for the lengthy period of time required for system specification [Gould and Lewis, 1985].

> The supplier is often asked to do "free design" and quote a price for satisfying a user's needs, without those needs being specified quantitatively in terms of performance requirements. In a cost competitive environment, less attention to detail results and critical flaws creep into the design [White, 1987, p. 25].

4.5.2.1 Who Manages Specifications?

The specification process is a system which develops another system [Hamilton and Zeldin, 1979, p. 29]. Who leads this process? This critical question is ignored in most systems literature. Little attention is given to the *management* of the process.

> It remains to be seen if the various bodies who interact in a system, those who use it, those that pay for it and those that build, maintain and evolve it, can cooperate to ensure that the systems have a maximum useful life span and that systems that are created are also subsequently controlled [Yavneh, 1985, p. 137].

4.5.2.2 Developer's Traditional Responsibilities

Traditional literature assumes that the developer is responsible for determining the user's requirements. The user is not viewed as the decision-maker, but simply as a repository of some information which is needed by the "real" decision-maker, the systems developer. The user becomes an object which is being evaluated by the developer. Most systems specification literature focuses on telling the developer "how best to extract the necessary detail about real user requirements" [Macro and Buxton, 1987, p. 52].

Traditionally, programmers interview users to determine their needs, then design systems to meet these needs. Once the coding phase of systems development begins, programmers interpret users' needs, decide how to prioritize users' needs and balance users' needs with technical requirements [Leventhal, 1986, p. 49].

As the impact of systems increases, users may feel that they have better understanding, but less control to fix problems [Bjorn-Andersen, 1985, p. 841].

4.5.2.3 Adding User Responsibility

Socio-technical theories of systems development give the user a more responsible role. As a participant in the process of defining needs and evaluating how the system is meeting these requirements, the user must understand the details of a system in order to control it.

4.5.2.3.1 Participation Not Sufficient

The practice of specification remains faulty when users and sponsors are assigned a participatory role in the process *yet remain unable to fully contribute*. Users and sponsors, overwhelmed by system complexity, are subtly placed in a subservient role. Technical priorities supercede the organization's needs. The expertise of users and sponsors is ignored, when developers' expertise dominates.

> Getting useful design information from prospective users is not just a matter of asking. Many users have never considered alternate or improved ways of performing their tasks and are unaware of the options available for a new design. Further, in trying to communicate, designers may unwittingly intimidate users, and users may unfortunately become unresponsive [Gould and Lewis, 1985, p. 303].

4.5.2.3.2 Understanding Needed

Until the users and sponsors understand how the system will interact within the organization, they are not in a position to fully contribute their knowlege and expertise.

> Participation is little more than hostage-taking, if the users are not given a training over and above the technical training traditionally offered by systems vendors. This training is first and foremost oriented towards appreciation and operation of the system. More elaborate training programs are called for in order that users may be trained to understand, evaluate and even design systems [Bjorn-Andersen, 1985, p. 845].

4.5.3 Real People Specify

Real, individual people specify systems. Real, individual people use systems. These people are alive, they have cultures, they have abilities, they have priorities, they have values. Some people like each other--others dislike each other. Some people understand each other--other people misunderstand each other.

4.5.3.1 Most Approaches Ignore People

Most approaches to systems development ignore the real, individual nature of each people. Traditional theories of systems development assume that, in a well-structured development, the individuality of participants and of their interactive relationships is irrelevant. Even when the systems development process is based on current tools, techniques and tasks, such as the CASE methodology (Computer Aided Software Engineering), there is a basic assumption that the primary task of systems development management is to *force adherence to the underlying task structure.*

4.5.3.2 <u>A Few Observers See Real People</u>

A few systems observers see real people specifying systems. These researchers question the assumption that people should and will conform to pre-determined technological goals.

> What is missing from the areas cited above (CASE tools, expert systems)? People. Software-development productivity will not just have a technological solution...the area of management strategies for CASE is an important partner to the software technology [Chikofsky and Rubenstein, 1988, p. 17].

4.5.4 <u>Collaboration Among Individuals</u>

Individuals who are technical experts *can* collaborate with individuals who are operational experts. The integrated systems of the 1990's require this collaboration.

4.5.4.1 <u>Supporting Operational Priorities</u>

An expert developer's strength lies in building technically superior systems, not in operational decisions. When developers place technical requirements above operational needs during the design of systems, the systems become ends in themselves. They fail because they are not designed to "support the realization and management of the real activities (the organizational processes)" [Essink, 1986, p. 57].

4.5.4.2 Recognizing Political Factors

The specification of system requirements is not simply a matter of judiciously selecting the most logically sound ways of meeting obvious needs. A system is the end product of political negotiations. An important factor during the development of quality systems within organizations is the successful fostering of productive, social innovation between very diverse groups.

4.5.4.2.1 Systems Research Focuses on Developer's Job

The focus of nearly all systems development research is on the jobs of system designers and programmers. The top goal of this research is to improve the work of the systems designers and programmers.

Negotiating with users makes the life of a systems designer more difficult. Programmers who must accommodate user priorities are frustrated with limitations on their technological flexibilities.

4.5.4.2.2 Successful Integrated Systems Also Focus on User's Job

Successful integrated systems merge information processing with operations. The design of these systems *requires that the focus is also on operations--the jobs of the users.*

Technological systems can enrich or weaken workers' jobs [Bjorn-Andersen, 1985]. Socio-technical theories emphasize the value of worker-determined organizational priorities.

In Europe there is an emphasis on democracy in the work place. Norway has federal laws which enforce the rights of workers for self-determination. Workers have the right to determine the impact of technology on their own jobs [Elden, 1986].

Some American businesses, attempting to bolster productivity and quality, incorporate worker-involvement in decision-making. Worker involvement becomes a critical factor within organizations who attempt to build integrated systems to improve productivity and quality.

> Change thrives where workers as well as managers are involved, where everyone realizes why the improvement is needed, participates in the design, understands what's required to operate it, and feels secure that management's real intentions are to help them help their company [Goddard, 1987, p. 67].

4.5.5 <u>Early Test Plan Scenarios Manage Collaboration</u>

Early test plan scenarios, developed prior to system coding, ensure that software code meets user needs [Bryan and Siegel, 1984, p. 64] by managing collaboration between users, sponsors and developers.

> "A formal education in computer science is not an adequate - not even an appropriate - background for those who must design and install large-scale computer systems in business environments." (Education - 1974) This statement was made by Mr. George Glaser, president of the American Federation of Information Processing Societies, Inc. [Wasley, 1975, p. 105].

4.5.5.1 <u>Social Interaction Crucial During Systems Specification</u>

The design of complex systems which change an organization, reflect an organization's political order and its patterns of communication

[Scacchi, 1983, p. 52]. The manner in which people interact, the complex "web of social arrangements", is a crucial element in defining the nature of a system [Scacchi, 1983, p. 58].

4.5.5.2 Out-of-Control Interaction Disrupts

When interaction among team members is out of control, it will disrupt a development. If team members do not have a single topic to focus upon, different viewpoints conflict and disrupt innovative work. Managing collaboration is difficult. If individuals are threatened by diversity, they may reject the views of others rather than use these different perspectives to increase the success of the entire team.

4.5.5.3 Successful Collaboration Improves

The paradox of collaboration between diverse people is that the same differences which produce discord can stimulate innovation. For collaboration to succeed, sponsors, users and developers need to overcome the separation caused by the boundaries of their separate groups, and redefine a new group, a group of collaborators, with a boundary which encompasses members of all three groups [Smith and Berg, 1987, p. 104].

If the new group becomes viable, another situation may occur. When the boundaries of the development team become strong, users, having learned new skills, may lose their appreciation for the difficulties the system presents for the novice user [Gould and Lewis, 1985, p. 303].

4.5.5.4 The Language of Collaboration

Collaboration among users and developers requires the use of a language which everyone understands *in the same manner*. Many researchers are trying to develop a "best" methodology for providing a unified model of a system which team members may use to evaluate the specifications and develop the system [Krasner, 1985, p. 22).

4.6 The Languages of Requirement Specifications

Specifications which focus on visible, external characteristics of a system describe activities a system must perform. Formal, abstract program specifications describe a system's internal mathematical relationships. Specifications of user needs describe properties which the system must exhibit.

4.6.1 Different Languages for Different Folks

Different communication formats are used for specifications. Natural language is used to specify user needs. Technical languages are used to specify developer's needs.

4.6.1.1 Ambiguity of Natural Language

Although user's familiarity with natural language improves the user's understanding, natural language hides ambiguous interpretations

[Poston, 1985, p. 63]. Users, sponsors and developers incorporate their personal viewpoints when interpreting words.

Ambiguity is insiduous when it remains hidden. When each participant is secure in his/her own interpretation, it becomes extremely difficult to identify the existence of ambiguity. Complex systems development requires unambiguous, formal specifications.

4.6.1.2 Formal Languages Hide Operational Blunders

Formal languages improve ambiguity but they hide operational blunders. Mathematical languages, notations and charts are designed to be unambiguous. Formal specifications may "lead the specifier to raise questions that might have remained unasked, and thus unanswered, in an informal approach" [Meyer, 1985, p. 22].

A problem with formal languages is that abstract, logical structures may appear so strong that reviewers assume they *must* support viable systems.

Rigorous mathematical languages are used to represent numeric programs, not to represent operational situations. There appears to be no language system which successfully represents complex systems [Krasner, 1985, p. 123].

4.6.2 This Thesis Suggests Using Several Languages

This thesis suggests using several languages. Different languages are approapriate for different system specification phases.

Natural language is appropriate for specifying objectives. Natural language, helped by tables and charts, is appropriate for analyzing functionality. Formal languages are required for specifying structure and design. The script language of scenarios is powerful when specifying system operations.

This thesis uses detailed script to develop test plan scenarios which uncover ambiguities (see Table 4.2). Scenarios integrate different system views into one understandable view.

Table 4.2 Contribution of Script to Specification Languages.

Capabilities of Traditional Language Classes

Class	Subject matter	Language	Semantic Capability
Requirements	The problem, user needs, environmental constraints	Natural language	Expert knowledge
System Specification	External, known characteristics of system	Notations from engineering, relational data bases, math models	Control state change, multi-processes, data relationships, performance measurements
Abstract program specification	An abstract operational description	Predicate calculus, recursive functions	Input/effect/output of operations, absract data types formal proofs

[Abbot and Moorhead, 1982, p. 298]

Contribution of Script Used in Early Test Plan Scenarios

Class	Subject matter	Language	Semantic Capability
Test Plan Scenarios	Active system, time sequences, simultaneous interaction, movement	Natural language script, layouts screenplays, action, diagrams tables, charts	Demonstrate interaction of system with people, systems, machines of environment

This scenario view lets users, sponsors and developers evaluate how the operations supported by the design [Boehm, 1984a, p. 81]. The concrete, simple actions used in scripts reduce the potential for ambuiguity and misinterpretation.

4.7 Improving the System

Test plans, developed from functional analysis and system structure documents, contribute two subtle improvements to the specification process. First, the system is presented in an operational context rather than a system development context. Second, users and sponsors have a responsibility to improve, rather than simply approve, the system specifications.

4.7.1 Transforming Operational Needs into Systems

The development process involves a series of transformations. People translate an operational situation into needs. People then envision solutions. These solutions are transformed into requirements, specifications, structural designs and code. Each transformation simultaneously changes the presentation format and the content of the planned system. As each transformation brings the planned system closer to the condition it will be in when it is operational, each transformation also travels further from the original operational environment. Users who are involved only in the final transformations are presented with a system which often is far removed from the original operational context [Sievert and Mizell, 1985, p. 56].

4.7.1.1 Developers Transform with Logic

The developers who accomplish these transformations focus on the rational logic of computer languages. Their vision of the system, their foundation for making decisions concerning the system, is based on this rational logic.

> As long as programmers use the current paradigm, which focuses on code, we will never achieve the orders-of-magnitude gain in productivity needed to end the software crisis [Sievert and Mizell, 1985, p. 57].

4.7.1.2 Users Transform with Operations

Early test plan scenarios shift the programmers' decision base from the rational logic of program code, to the operational functionality of test plans. Once a system is operational, users and sponsors are able to identify areas in which the system can be made more useable.

The development of complex systems within organizations is a process of social innovation. Although some observers of the development process still assume that systems development is best performed by a few experts aided by sophisticated techniques and computerized tools, many current researchers recognize the importance of sponsor and user contribution during the development process.

4.7.1.3 Developers' Logic Overwhelms Users' Operations

Most development literature ignores the power exercised by developers over users and sponsors. Whether this power is openly or subtly exercised, it interferes with the full exploitation of user and sponsor expertise during systems development. When developers interview users, when they make hardware decisions, when they determine software selection, they are limiting the opportunities for contribution by users and sponsors. The exercise of power by developers can diminish the users' contributions [Markus and Bjorn-Andersen, 1987].

4.7.1.4 Unique Nature of Developer-User Communication

Software engineering is significantly different from other forms of engineering because of the nature of the communication between users and developers which is required during the development of systems [Macro and Buxton, 1987, p. 27]. The power of developers, during initial phases, resides in "tentative acts of influence" by which developers control the content and format of communication between users, sponsors and developers [Macro and Buxton, 1987, p. 27].

4.7.1.5 Controlling Transformation

Controlling the transformation of operational needs into technological systems becomes increasingly difficult with integrated systems. A rapidly escalating problem in the development of complex systems is the maintenance of "intellectual control over the development of large software systems that exist in many versions (through time) and many configurations (concurrently)" [Shaw, 1985, p.

215]. Intellectual control over large systems requires the collaboration of many people.

4.7.1.6 Managed Collaboration Controls Transformation

Collaborative innovation requires "group exploration" [LeFevre, 1987]. Individuals must be able to access information known by other collaborators. A management structure must exist to enable collaboration, to provide a cohesive structure for the work and to permit evaluation of the development process. A single model must be available to all collaborators. A control process must be established which allows this single model to be used by all collaborators and controls the methods by which this model is modified by the collaborators.

The use of early test plan scenarios manages this collaboration by providing the single, consistent model on which the collaborators can focus their activities [Richter, 1985, p. 193].

4.7.2 Evaluating the Usefulness of a Specified System

Traditionally, developers themselves evaluate how well their designs fulfill requirements [Potosnak, 1987, p. 91]. When developers evalute the usefulness of a system design, there is an assumption that the design of a system merely transforms specifications into programs [Branstad and Powell, 1984, p. 75].

4.7.2.1 Developers' Freedom to Evaluate Design

Developers may insist that they have absolute freedom in design evaluation, arguing that *how* a system is developed does not change *what* is developed. Some argue that developers must be free from "user interference" during the design and coding of a system.

> Including design information in requirements specification is wrong because it restricts design trade-offs. If you tell a person what is required in a product as well as how to build it, you have specified requirements *and* designed the product. Design information is sometimes considered extra rather than wrong information, but however it is classified, it doesn't belong with requirements [Poston, September 1987, p. 83].

4.7.2.2 *How* Affects *What*

Particularly with integrated systems which merge information technology and operations, *how* a system is designed affects *what* the system does operationally. The use of early test plan scenarios gives users and sponsors the opportunity of assessing the operational impact of a system's design.

4.7.2.3 Collaboration Improves the *How* and the *What*

A function which is operationally simple may be technically complex [Roman, 1985, p. 15]. When there is no user or sponsor review of functionality after it is affected by design, programmers make decisions concerning the inclusion, modification or omission of functionality. Programmers, not usually qualified to make business-oriented decisions, use system-oriented priorities [Leventhal, 1986, p. 49]. Decisions concerning useability are made by programmers whenever the system's useability is not explicitly stated in operationally measureable terms prior to systems coding.

Design development opens new possibilities for functionality. When developers, users and sponsors review the functionality, after the design has been established, new possibilities are available for improving the system at the most economical stage -- prior to coding.

> We recommend that potential users become part of the design team when their perspectives can have the most influence, rather than using them post hoc as part of an "analysis team (of) end user representatives" [Gould and Lewis, 1985, p. 302].

4.7.3 Early Test Plan Scenarios Customize Systems

Prototyping has emerged as a powerful tool in early customization of system-operator interface characteristics. When developers and users collaborate to develop screen displays and operational sequences, the system's functionality is being customized to fit the unique characteristics of the users' environment.

Early test plan scenarios continue this customization process by providing a vehicle for user-sponsor-developer collaboration for the entire system scenario. Collaboration on forms, screens, and individual programs are necessary, but insufficient. User-sponsor-developer collaboration activities for complex systems must address the system's interaction with people, machines and other systems (see Table 4.3). Test plans help customize the system. Test plans extend user-developer collaboration beyond forms and prototypes.

Table 4.3 Early Test Plan Scenarios Customize Systems.

Customization Product	*Process - Type of Interaction Being Evaluated*
Forms	*Customization of forms for users and sponsors.* Interaction of new system's inputs & outputs with users.
Prototypes	*Customization of operator screens for users.* Interaction of new system with users.
Test Plan	*Customization of system for organization.* Interaction of new system with users, machines, existing systems, temporal and spatial conditions of operating environment.

4.7.4 Promoting System Quality

Quality standards improve requirement specifications. Although researchers develop and promote standards to improve the systems development process, only those methods, structures and terminologies which are tried, found useful and adopted by developers are actually accepted as standards [Poston, 1985, p. 64]. Some researchers claim that the development process is so dependent upon the domain of the development situation, that no universal standards can be developed [Blum, 1985, p. 21].

4.7.4.1 Different Quality Criteria

Different participants in the development process may use different criteria to measure the quality of a specification. Users may be concerned about a system's responsiveness, its risks to operations, and the flexibility they will have to modify the system. Sponsors may be concerned about the final impact of the system, meeting schedules, the total cost of the development, and the compatibility of the system with other systems. Developers view quality by the feasibility of the system, the simplicity of the system, the efficiency of the system and the achievability of the system within hardware, time and resource constraints [Evans and Marciniak, 1987].

4.7.4.2 Tools Improve Quality

As the cost of developments escalate, there is a proliferation of technical tools to promote quality. These tools are attractive to developers who want to improve their productivity and to managers who find comfort in their apparent usefulness. Tools automate many areas of systems development: specification, analysis, prototyping, design, programming, testing, verification, documentation, project managment, measurement of quality [Cavano and LaMonica, 1987, p. 30].

4.7.4.3 Statistics Improve Quality

Statistical certification of software reliability can evaluate the cohesiveness of the software as development progresses, determining whether changes and additions are helping or destroying the structural soundness of a system [Mills, Dyer and Linger, 1987].

When statistical techniques are "tools", used to increase the logical correctness of coding and design, they may improve the development. When statistical techniques replace human judgment, they prevent users and sponsors from contributing their expertise to the development. While tools abound in the academic community, there are still no widely accepted tools used in industry [Kishida, Teramoto, Torii, Urano, 1987, p. 11].

4.7.4.4 Problems with Tools and Statistics

Subtle problems may exist with these tools. The creative process is complex. We do not know whether "mundane" tasks waste time, or whether they actually contribute to performance by forcing a deliberate slowdown during which the developer is able to gain new insights into the system's development.

4.7.4.4.1 Tools May Discourage Innovation

Efficient tools may discourage innovation and improvement. People tend to accept their value without questioning their impact. When developers have sophisticated tools, it may be harder for users and sponsors to challenge the impact of a design.

> Future environment tools will analyze data as development progresses, providing control early on. Entire systems will be evaluated and potential software-quality problems will be identified as quality analyzers predict and assess a project's cost, schedule and quality at each development stage [Cavano and LaMonica, 1987, p. 32].

4.7.4.4.2 <u>Quality May Be Assumed</u>

The parable of the emperor's new clothes applies to development situations. If tailors state that new clothes have quality, they acquire "quality." When systems designers state that a system design has quality, particularly when sophisticated tools are used to "ensure" that quality, users and sponsors may be lulled into agreeing with the developers and accepting the "truth" of the tools. Only one little child trusted himself enough to remain unfazed by the words of the tailor. The adults eagerly accepted the tailors' pronouncements.

A similar situation can occur when well-paid professionals develop systems. Most users and sponsors accept the developers' conclusions. The occasional doubter, the "child" who questions, may be dismissed as being well-meaning, but ignorant [Eastman, 1986]. Only when attempts are made to activate the system, do the sponsors and users recognize that the system has serious faults -- that the emperor is really naked.

4.7.4.5 <u>Discipline and Front-End Planning Improves Quality</u>

People who will build integrated systems in the 1990's will not have the luxury of experimenting with quality. The Department of Defense places improving the quality of systems development third in its list of critical priorities [Norman, 1989, p. 1543].

Extensive research sponsored by the Department of defense has found that a disciplined development environment and front-end planning are essential for systems quality. *Early test plans were used by every* *single developer recognized by this research as having established a record of developing quality systems* [Humphrey, 1988].

4.8 Knowing When to Stop Specifying and Start Coding

A difficult problem with requirement specifications is determining when the specifications are "good" enough to proceed to the next phase of development. Most specification methodologies fail to explicitly state how the quality of the specifications will be measured.

4.8.1 Defining When Specifications Are Complete

Basili's research on software development supports a development process which *begins* with explicit definitions of quality and then considers which process to use to achieve that measure of quality [Basili, 1988b]. Quality objectives must be defined operationally "relatively to the customer, project and organization" [Basili, 1988a, p. 88]. System development which focuses on quality requires additional, rigorous effort, planning and collaboration. With the cost of maintenance, the prevalence of software failures, and the increasing complexity of hardware and software, there may be no other choice.

4.8.2 Complete Specifications Identify Start and End Point

The specification provides both the starting and ending point for the developer. The developer uses the specification as the basis for developing the structure and content of the system design. During development, the specification should offer guidelines to the developer for most decisions affecting the operational characteristics of the system. Finally, the developer validates the completed system by comparing it with the specification of the intended behavior of the system [DeMillo, McCracken, Martin, Passafiume, 1987, p. 3]. Explicit measureability of the useability of a system's operations should be a fundamental part of specification "quality". Yet usually in systems development, operational tests are written, reviewed and used only after most, or all, of software coding is complete [DeMillo, McCracken, Martin, Passafiume, 1987, p. 23].

4.8.3 Complete Specifications Completely Describe Operations

Early test plan scenarios support quality with explicit, measureable demonstrations of the operational useability of the system. The specific, measureable functionality of a system demonstrated in test plans supplies specific system objectives to determine when the specification is "good enough" to proceed.

When useability is not defined using quantitatively measureable terms, "less attention to detail results and critical flaws creep into the design" [White, 1987, p. 25]. Early test plan scenarios translates general objectives into specific, measurable terms. Concrete, measureable standards for useability provide the specific objectives which improve the software development process and the final system [Potosnak, 1988, p. 89].

4.8.4 Agreement that Specifications Are Complete

A primary function of managment is to ensure that decisions are made by the right people, using the right criteria. The decision that specifications are complete must be made by both users and developers. The criteria must include both technical and operational factors.

4.8.4.1 Reviews to Promote Consensus

A valuable management tool to enable collaboration and promote consensus is the formal review process. Formal reviews reduce ambiguity and decrease misunderstandings, omissions and discrepancies which occur during complex developments [Parnas and Weiss, 1987, p. 259]. The participation of team members is necessary to rigorously challenge the specification. The purpose of these reviews is to identify problems with the specifications which will decrease or prevent the system's operational useability.

> What is required is a usability test, not a selling job. People who have developed a system think differently about its use, do not make the same mistakes and use it differently from novices [Gould and Lewis, 1985, p. 302].

4.8.4.2 Reviews Improved when Preparation Required

Time can be saved, and the quality of the review improved, when individuals make written comments after their initial review of a document, and these comments are consolidated by a person who is responsible for the acquisition and categorization of these comments. Non-controversial errors are corrected prior to the team review session and notification is sent to all team members. During the review, team members focus on those problem areas with controversial resolutions [Fryer, 1985, p. 67].

Reviewing system requirements which have been influenced by the design, prior to coding, focuses attention on particular aspects of the functionality. Delegating responsibility for the useability of a system reduces errors and improves development quality [Parnas and Weiss, 1987, p. 264].

4.8.5 Improving Systems Requires Individual Responsibility

Individual responsibility, of users and developers, improves the specification process and the final system. Early test plan scenarios promote individual responsibility.

Test plans require individual and team scrutiny of the system's design. Some developers who try to involve others in the requirement specification process become frustrated by the apparent lack of concern and commitment by users and sponsors to this process. Users and sponsors may argue that it will be easier for them to evaluate the system after it is "completed."

When users and sponsors do not "own" development problems, their energy is focused on complaining about the development rather than on solving problems. "Taking the risk to own or be responsible for the solution to a problem is a necessary prerequisite for an environment of effective problem solving" [Slaughter, 1980, p. 17].

4.9 Cost Justification of Early Test Plan Scenarios

Sponsors of projects are concerned about excessive time demands on their employees if the employees are to participate in systems development [Gould and Lewis, 1985, p. 304]. More research is needed to measure the cost savings which occurs with improved user participation.

4.9.1 Cost of Poorly Defined Systems

The cost and time involved in iteration, negotiation and review of specifications needs support from additional case study and research work to convince sponsors of the ultimate value of such activities [Buckley, 1984, p. 39]. There is little research about methods to validate functional specifications [Chandrasekhara, Dasarathy and Kishimoto, 1985, p. 71]. Much more work is needed to develop techniques to understand, test and evaluate complex systems.

> When complicated software is combined with intricate electronic machines, elaborate communications systems and the quirks of users, a typical computer system represents an exceedingly high level of complexity. Such systems can fail in many different and unexpected ways. "Simply put, modern-day, complex mechanisms do not work properly," Morris (Robert Morris, chief scientist at the National Security Agency) says. "We have to learn to cope with complexity" [Peterson, 1988, p. 199].

4.9.2 Reducing Total Costs with Early Test Plan Scenarios

Total costs, over the entire life cycle of a system, are reduced when test plan scenarios are developed before system coding. When presented with a completed system, users often find that the system does not perform as they had expected. Even when specifications are carefully prepared, the users' interpretations of these formal specifications are often critically different from the interpretations of the system developers.

> *Getting concise requirements to design a system and a method to document them that is understandable to both the user and development technicians is the difficulty [Zeimetz, 1988, p. 160].*

4.9.2.1 Early Test Plan Scenarios Improve Process

The *process* used to create complex systems may be as error-prone as the systems themselves [Poston, March 1988, p. 86]. Before the system becomes operational, the primary method of uncovering its errors is through software testing. When the process used to produce these tests is faulty, the tests themselves can create additional problems for the development process. When the process of creating tests has problems, these problems are compounded in the final system. The process of creating tests involves defining what to test, when to test and how to test. Important questions concerning the test plan process are: *when* to develop the test, *who* should develop the test and *how* to evaluate the test plans.

> If developers know how the product will be tested before coding begins, they are likely to design out the most-probable errors [Poston and Bruen, 1987, p. 59].

4.9.2.2 Early Test Plan Scenarios Effectively Allocate Costs

Early test plan scenarios effectively allocated costs by prioritizing system functionality. A frequent error when designing tests is to ignore critical functionality or to redundantly retest the same functionality [Poston, March 1988, p. 86]. Early test plan scenarios link each portion of the test to a specific section of the functional analysis, ensuring that all sections of the functional analysis are tested.

The process of developing functional tests can be improved by closely matching the test environment to the user's operational environment, having the user participate in test development, and including expected problem situations in the tests. [Poston, March 1988, p. 86].

> There is no denying that (early) user testing still has a price. It is nowhere near as high as is commonly supposed, however, and it is a mistake to imagine that one can save by not paying this price. User testing will happen anyway: If it is not done in the developer's lab, it will be done in the customer's office [Gould and Lewis, 1985, p. 306].

Chapter 5

FRONT-END PLANNING WITH EARLY TEST PLAN SCENARIOS IMPROVES OTHER DEVELOPMENT METHODOLOGIES

Long before the program editor is invoked, substantial time must be invested in painstaking up-front planning. Frankly, planning is not much fun, and without good analytical tools, developers are easily tempted to skip on this stage or skip it entirely. Planning certainly does not deliver the short-term feedback that comes from stepwise development -- adding code incrementally, testing it and seeing the new code run to completion.
[Anderson, J., 1988, p. 9]

5.1 <u>Developers Avoid Planning</u>

Although most researchers and practitioners acknowledge that inadequate requirement specifications are the primary cause of system development failure, software developers tend to avoid planning. It is "easier" for users to identify problems in operating software than to uncover ambiguities, inaccuracies and deficiencies in abstract functional specifications and designs [Poston, August 1988, p. 2]. Developers get paid for their work, whether it is before or after a system becomes operational.

5.1.1 <u>Traditional Development Methods Start *At Coding*</u>

Traditional development methodologies and tools focus on the later phases of the development process, helping with coding, program testing and debugging [Anderson, J., 1988, p. 9]. Software development rarely follows the basic rule of quality control--find and fix blunders early.

5.1.2 Early Test Plan Scenarios Ensure Quality *Before Coding*

The front-end planning approach presented in this thesis improves system quality at the most effective--and least costly time--before program coding. In particular, early test plan scenarios ensure that all design blunders which *can* be identified, *are* identified--*and corrected*--*before* a system is concretized in code. Early test plan scenarios improve the specification process by making the requirements themselves more "physical" [Poston, August, 1988, p. 2].

5.2 Comparing System Development Methodologies

This thesis argues that systems development is improved when developers include front-end planning with early test plan scenarios in their development methodology. Development approaches may be organized into six different categories: (1) Systems Development Life Cycle, (2) Traditional/Classical, (3) Structured, (4) Automated, (5) Prototyping, (6) Information Center. Front-end planning with early test plan scenarios can be adapted to improve any of these six approaches.

5.2.1 Five Comparison Questions

The front-end planning approach using early test plan scenarios may be compared to other methodologies by answering five questions which reveal underlying assumptions fundamental to each methodology (see Table 5.1).

Table 5.1 Five Comparison Questions.

1. Who participates in the specification process?
Front-end planning with early test plan scenarios promotes collaborative innovation among sponsors, users and developers.

2. How should the specification process be structured?
The four phases (in front-end planning with early test plan scenarios) are stating: (1) Objectives, (2) Analyzing functional needs, (3) Designing the system structure, and (4) Developing operational test plan scenarios.

3. What terminology is used to document the specifications?
Front-end planning with early test plan scenarios uses different formats for different phases. Natural language is used to state objectives and functional needs. Structured languages and formal mathematical terms are used to define a system's structure. Script is used to represent the activated operations of the system in measureble terms.

4. How are components of successive phases linked together?
Front-end planning with early test plan scenarios uses a rigorous numbering system to link components of a single phase with each other and with corresponding components of other phases.

5. How is quality in the specifications defined and measured?
Early test plan scenarios, developed prior to coding, are the explicit, quantitative standards which are used to evaluate the useability of the requirement specifications and to later determine the "goodness" and "completeness" of the final system.

5.2.2 Six Categories of Specification Methodologies

Current system specification approaches may be assigned to one of six major categories (see Table 5.2). Organizations frequently consolidate several approaches into a single development process. Front-end planning wity early test plan scenarios fits within the Systems Development Life Cycle (SDLC) approach, yet can incorporate approaches from each of the other five categories. Most developments utilize a variety of specification methods [Necco, Gordon and Tsai, 1987, p. 466]

Table 5.2 Approaches to Develop Information Systems.

Approach	Number Using	Percent Using (n=97)	Number Considering Using	Percent Considering Using (n=non users)
* Systems Development Life Cycle	67	69%	2	7%
* Traditional/Classical	67	69%	4	13%
* Structured	60	62%	8	22%
* Automated	56	58%	22	54%
* Prototyping	45	46%	15	29%
* Information Center	73	75%	11	46%

[Necco, Gordon and Tsai, 1987, p. 466]

5.3 Selecting One Process from Each Method

Many variations of specifications approaches are represented in each category. To compare the front-end planning with early test plan scenario process with other methodologies, one representative process is selected from each category (see Table 5.3).

Table 5.3 Representative Methods of Specification.

CATEGORY	REPRESENTATIVE PROCESS
1. Traditional/Classical	Life Cycle Method
2. Systems Development Life Cycle	Jensen, Tonies Methodology
3. Structured	SADT (Structured Analysis Design Technique)
4. Automated	CASE (Computer Aided System Engineering)
5. Prototyping	Scharer Prototype Methodology
6. Information Center	RML (Requirements Model)

5.3.1 The Traditional/Classical Approach: Traditional Life Cycle

During the early 1950's, systems analysts and operations researchers were usually engineers. They patterned the traditional/ classical approach after their mechanical training. The first major systems development method, the Life Cycle approach, emerged from this tradition. It phases systems study into preliminary design, detailed design and programming/installation (see Table 5.4). In 1963, Laden and Gildersleeve expanded this basic structure to include a feasibility study, objective definition, system design, programming, filemaking, documentation and testing [Agresti, 1986, p. 4].

The traditional/classical approach views systems development as an engineering task, *emphasizing the expertise of the developer.* Front-end planning with early test plan scenarios views systems develoement as collaborative innovation, which *emphasizes the expertise of the user and sponsor as well as the expertise of the developer.* Traditional life-cycle models focus on design and coding [Boehm, 1976] with less emphasis on the specification phase.

Front-end planning with early test plan scenarios uses a variety of formats; the traditional/classical approach uses first-generation tools, natural lanuage descriptions, flowcharts and file layouts to describe the requirements of a proposed system [Necco, Gordon and Tsai, 1987, p. 462]. The traditional approach assumes an evolution of one phase into another phase and does not require the component linkage control promoted in this thesis. Quality in front-end planning with early test plan scenarios is evaluated by the useability; quality in the life cycle process is measured by adherence a rigid schedule.

Table 5.4 The Conventional Life Cycle Approach.

Phase 1:	*Specification.* A statement of "what" the software will do, following a detailed analysis of the requirements.
Phase 2:	*Design.* "How" the software will meet the requirements; the structure of software units that perform specified functions.
Phase 3:	*Code.* Implementation of the design in a programming language.
Phase 4:	*Test.* Verification that the code executes without failure, and validation that the completed software is acceptable to the users.

[Agresti, 1986, p. 2]

5.3.2 Systems Development Life Cycle Approach (SDLC): Jensen-Tonies Methodology

The SDLC approach develops specific phases, tasks and products to organize the systems development process. Early SDLC approaches expanded the Life Cycle methodology by assigning specific tasks, products and tools to each phase. SDLC methods often integrate processes from other approaches such as the structured, automated and prototyping methods [Necco, Gordon and Tsai, 1987, p. 463].

While SDLC methods retain the basic theme of the traditional Life Cycle approach, the Jensen-Tonies methodology limits the design phase to only conceptual design, incorporating the structural design with the code phase, and adds an operations and maintenance phase to the end of the life cycle (see Table 5.5).

Table 5.5 The Jensen-Tonies Methodology

Phase 1: Research.
Required operational capability document; engineering studies; experimental development activities.

Phase 2: Conceptual Phase.
Concept evaluation; systems requirements review; system decomposition & allocation; system design review; software requirements.

Phase 3: Design & Development Phase.
preliminary software2 analysis & design; preliminary design review; detailed software analysis & design; critical design review; coding and debug.

Phase 4: Software Code Testing.
Software subsystem integration & testing; software system integration & testing; software/hardware integration & testing; system performance testing; functional configuration audit; acceptance test & evaluation; customer delivery & physical configuration audit; operational testing & evaluation; operation & maintenance.

[Agresti, 1986, p. 3].

The SDLC approach highlights the role of the developers during system development. As with traditional approaches, most SDLC methodologies focus on coding a system. A major weakness of current SDLC practices is that less than 15% of development time is allocated to defining requirement specifications. In their hurry to begin designing and coding a system, *33% of SDLC developers do not specify requirements.* [Necco, Gordon, Tsai, 1987, p. 68].

Front-end planning with early test plan scenarios and SDLC both promote an organized, sequential approach to systems development that describes how to measure the completion of each phase. They both define products and activities that must be completed before one phase ends and another begins. Front-end planning with early test plan scenarios requires that components of one phase are explicitly linked to components of other phases; SDLC methods have no standard linkage requirement. The quality of software developed with front-end planning with early test plan scenarios is measured quantitatively. Traditional SDLC methods define and measure quality by the successful completion of each task and activity associated with the specification phase.

5.3.3 The Structured Approach: SADT (Structured Analysis and Design Technique)

SADT^{tm} is a systems description language and methodology which was developed by Douglas Ross (of MIT), in the late 1950's [Marca and McGowan, 1988]. It has been used to model systems, particularly the requirement specification of systems, for thousands of commercial developments (see Table 5.6). Under the name of IDEFO, (Integrated Computer-Aided Manufacturing Program), SADT is a standard for the U.S Department of Defense and has been used in hundreds of military and industrial systems developments.

SADT may model any system. It reveals, with a single viewpoint, the overall structure of a system and the interrelationships of all system components. SADT improves user-developer collaboration with an author-reader cycle. SADT *requires* the contribution of user expertise.

Recognizing that users have limited time to devote to a project, SADT incorporates a review/reiteration process which tries to maximize user-developer cooperation by maximizing user understanding and feedback while minimizing user time requirments.

Table 5.6 Seven Fundamental Concepts of SADT.

(1) SADT attacks a problem by building a model or a representation of the problem...Multiple SADT models of a system from different viewpoints may be required for complete understanding.

(2) Analysis of any problem is top-down, modular, hierarchic and structured.

(3) SADT differentiates...between the creation first of a functional model...and then the creation of a design model.

(4) SADT models both things and happenings. The complete SADT model of a problem must show both aspects properly related.

(5) The language of SADT is a diagramming technique which shows component parts, interrelationships between them, and how they fit into a hierarchic structure.

(6) SADT methods support disciplined, coordinated teamwork, which is required in order to produce results which reflect the best thinking of a team.

(7) SADT methods require that all analysis and design decisions and comments thereon be in written form.

[Ross, 1976, p. 2.1].

SADT is used to understand and represent a system through a set of diagrams which fit together to form a model with a single view of a system. Several complementary models may be created for a single system, each with a unique viewpoint. Each model should represent only a single viewpoint for the SADT method to accurately model a system.

Most complex systems developments using the SADT methodology have one complete SADT model to represent the system from the data flow viewpoint and one complete model from a procedural viewpoint.

A major strength of SADT is that it forces the definition of the boundary between the system and the non-system world. SADT then forces the definition of boundaries and hierarchical relationships between each of the system's subcomponents. This rigorous structuring of the system establishes a solid foundation for the later concretization by coding portion of the development.

Rigorously structured SADT diagrams facilitate post-operational modifications. When the basic system structure is clear and the relationship of the components is firmly established, it is easier for analysts to determine where maintenance modifications are needed and how best to fit these changes into the tightly interrelated system.

SADT can incorporate automated CASE tools. But automation is not critical for successful use of SADT. Douglas Ross, the originator of SADT, still prefers to construct SADT diagrams with pencil and paper.

SADT can be successfully integrated into front-end planning with early test plan scenarios. The initial development of researched for this thesis used SADT to create two complete models of the complex development. One model traced the flow of data and one model represented the interaction of procedural components.

SADT's emphasis on user-developer collaboration was a major influence during the during this research. SADT uses explicit, clear diagrams as the focus of communication among participants in the development process. This approach underpinning this thesis borrows the concept of a focal point for collaboration from SADT. Each phase of this thesis's approach uses specific documents to focus the collaborative activities.

SADT's process links all subcomponents into a hierarchical scheme. Front-end planning with early test plan scenarios uses this linkage scheme and further develops it to link together the components of other phases of the development process.

SADT is a powerful tool for structuring a system. It is not meant to operationalize or measure a system. A critical premise of the front-end planning with early test plan scenarios methodology is that there are *different* aspects of specifying a system, and *each* aspect is best accomplished by using *specific tools.* The best tools for one aspect are often not appropriate for another aspect.

The development of complex systems is hampered by attempts to make a *particular* tool the *universal* tool. Front-end planning with early test plan scenarios does *not require* specific tools for any phase of the specification process.

The selection of the particular tools to be used within each phase should be based on the requirements of the phase, the development conditions, the capabilities of currently available tools, and the developer's preferences.

5.3.4 The Automated Approach: CASE (Computer Aided System Engineering)

The CASE (Computer Aided System Engineering) methodology uses computer tools to structure user requirements and to design the system from the top down. Starting with the highest level design, CASE tools help developers automate design decomposition from the highest macro level into detailed sub-divisions [Anderson, J., 1988, p. 9].

CASE tools are at the forefront of those developments which attempt to "forge a creative partnership" [Yourdon, 1988, p. 740] between people and automation. The CASE based process is designed to facilitate user-developer communication, by providing descriptions of the system in several different formats including flow charts and structured diagrams. Although CASE developers advocate helping users understand the system structure, the developer remains the expert extracting information from the user [Martin, 1988, p. 3]. CASE methodologies use a top-down approach implemented with one or more computerized tools to develop the system structure.

The most prominent variations of the CASE approach base their top-down design strategies on data flow, information interaction or operational procedures. While all three of these variations may be integrated into a single development, the CASE tools which support procedural analysis may be most directly used by front-end planning with early test plan scenarios.

The first development researched for this thesis included both data and information analyses which supported, and were integrated into, the procedural analysis. CASE includes a variety of tools which could be useful when using front-end planning with early test plan scenarios to develop a complex system. While some developers feel there remains an advantage to manually diagramming a system, others prefer to use CASE tools to automate the pictorial representation of a system.

CASE tools can be used to simulate screens and to generate reports which can be incorporated into the operational test plans. CASE tools for data dictionaries, project management information, syntax checking and documentation generation attempt to computerize some of the time-consuming activities which occur during complex systems development.

Although CASE tools claim to be total development methodologies, their actual role normally begins after the developer and user have defined the objectives and scope of the system [Topper, 1988, p. 71].

CASE tools are not used to define basic organizational objectives of the system development. While some tools claim to help develop the functional analysis, their primary role during this process appears to be that of documenting the work which is accomplished by people.

CASE tools could be used during the structural design of the system. Tools to decompose a system, to computerize the process of diagramming a system's structure, may increase the productivity of a developer.

The developer must continue to participate in the structural analysis, *to ensure that the process is sensible.* It is easy to become so infatuated with CASE tools, that the hard thought and judgment involved in structural analysis is bypassed.

A modification of currently available CASE tools would be particularly useful in controlling the linkages between different components of different iterations of the system.

There is no explicit function within CASE to measure quality of specifications by evaluating the useability of the system. Proponents of CASE argue that users are able to evaluate a system by understanding the diagrams and pictorial representations of the system which can be produced with CASE tools. CASE methods have no formal measurement criteria for the final system. Early test plan scenarios exercise the proposed system within the spatial, temporal and physical conditions of its future operational environment. CASE tools do not evaluate the useability of specifications. Front-end planning with early test plan scenarios establishes operational test plans as the official, formally approved measurement criteria to determine when the system has been adequately developed and should be accepted.

In 1984, Robert Poston, of Programming Environments Inc., began the development of an automatic test generator (which belongs within the CASE category). Funded by grants from the Small Business Innovative Research Program and the US Department of Defense, PEI was supported through three phases of development. Phase I, the feasibility study, was simple because PEI already had a working version of the test generator. Phase II ported the system to Unix and VMS systems. Phase III assisted in the commercialization of the system [Poston, January, 1988, p. 1]. **T** includes functionality to measure the quality of software, to determine when it is "good enough." Testing comprehensiveness is measured through formulas which determine the ratio of what *has been* demonstrated to what *should be* demonstrated [Poston, April 1988, p. 3].

T has been designed as a specification tool to generate test cases which measure the useability of specifications before any coding begins. Developers of **T** consider its most effective impact to be the capturing and communication of the *physicalness* of the system, which is easier to understand than requirement statements or structural diagrams [Poston, August, 1988, p. 2].

Users of **T** include the US Army and AT&T. By developing test cases, prior to testing, users have measured an 85% reduction in testing time, a 47% reduction in programmer-discovered bugs and a 95% reduction in customer-discovered bugs. By incorporating test plans into the specification process, users of **T** have moved a significant portion of system maintenance to the less costly front end of systems development.

It is likely that historians in the future, looking back on the extraordinary evolution of computing, will say that the true computer revolution was the one that automated the processes of design and programming. Manual structured techniques were a necessary preliminary to the automation but were not by themselves the true revolution [Yourdon, 1988, pp. 742-743].

5.3.5 The Prototyping Approach: Scharer Methodology

A prototype is an early version of a program that exhibits features of an operational system [Alavi, 1984, p. 556]. Prototyping creates a working system to help users define their needs and help developers understand users' needs [Taylor and Standish, 1984, p. 39]. It is based on a brief statement of user needs followed by a rapidly produced, operational model of a program which addresses these needs (see Table 5.7). Fourth generation languages and prototype languages such as PSDL are used to produce the prototype [Luqi, Ketabchi, 1988, p. 66]. Technologies are being developed to automate the transformation of a prototype program into an operational version of a program [Balzer, Green, and Cheatham, 1983].

The prototype approach which is successful for single programs and simple systems has limited use for complex developments. It gives no indication of how a system will react when it is stressed by large volumes of material or data, or by tiem and sequences. It does not evaluate the interaction of programs or systems. Components are not formally linked together but are simply transformed into new versions [Balzar, Green and Cheatham, 1983, p. 40].

Table 5.7 Four Phases of Prototyping.

Phase 1: Preliminary fact-finding.
Study business problem;
Generate alternatives.

Phase 2: Pregeneration Design.
Identify and define data elements;
Define "flat file" arrangement of data; Design high-level system flow; Describe procedural responsibilities; List transactions by type and effect; List reports, inquiries, updates.

Phase 3: Prototype Generation.
Task; Generate baseline prototype

Phase 4: Prototype Refinement Interactions.
Demonstrate prototype to users and record user comments;
Add to functional scope of prototype;
Supplement physical model with written descriptions

[Scharer, 1983, p. 37].

While front-end planning with early test plan scenarios requires the development of operational test plans to define the measurement criteria of a system before a system is coded, *prototyping is a process which eliminates testing.* [Balzar, Cheatham and Green, 1983, p. 40]. The risks of failure in a complex systems development increase when there are no quantitative measurement tests. *An agreement to "produce a quality system which meets user needs" is an agreement to disagree during the final assessment of the quality and of the useability of an activated system.*

Front-end planning with early test plan scenarios can use prototyping during the development of requirements for specific portions of complex systems, but not to represent an entire system.

Prototypes can simulate user-terminal interfaces successfully enough to enable viable user evaluation on a proposed function. However, the evaluation of complex systems requires a broad merging of temporal, spatial, informational and environmental factors which are not amenable to prototyping. While parts may be evaluated with prototypes, the critical aspect of complex systems which must be tested is the interaction of the parts with each other and with other entities in the operational environment.

5.3.6 Information Center: RML (Requirements Model)

Information center approaches to systems development were first used by IBM Canada to reduce system development backlogs by involving users in the process of defining and modifying computer applications [Necco, Gordon, Tsai, 1987]. Specialized languages are being developed to direct users in formally stating the "information" which should be represented in a system's database. Information-based systems developed attempts to be self-building. As users employ the information represention language to model the system and the environment in which it will operate, the system's foundation is defined and constructed.

Many researchers consider the management of information to be the weakest link in the development of complex systems. They claim that NASA's failures are directly linked to faulty information management which results in incompatible systems, delays of years in transfer of some information, and the lack of an overall system architecture to coordinate the interaction of critical space systems.

> The effects of bad information management permeate and pollute every level of NASA, driving away the best scientists and discouraging new recruits...NASA critics point to (faulty) information systems as a prime culprit in the agency's inability to exploit space....(NASA's) haphazard approach to computing in which incompatible systems isolate scientists from their fellow workers [Stapleton and Puttre, 1988, p. 8].

RML was initially designed to be a stand-alone tool for specifying requirements. (Borgida, Greenspan, Mylopoulos, 1985). Information methodologies are not able to evaluate the differences between the real world and the modeled world [Borgida, Greenspan, Mylopoulos, 1985]. While the model may contain some of the relationships which exist in the real world, it can never contain all of the relationships which exist. If a model were to exactly replicate the real world, the model would be the real world. Information methodologies can not yet determine when the information model is "good enough" to be useful.

> It is not that such conclusions were necessarily wrong; more to the point is that one could not know them to be correct. The problem arises when taking them for granted results in overlooking a different line of inquiry that might have led in a more meaningful direction [Eberhart, 1988, 72].

The improvement of software development practices requires a methodology which increases productivity while increasing reliability and adaptability of software. (Boehm, Standish, 1983, p. 30]. Development methodologies need to smoothly integrate all important facets of development.

> We are convinced that success in combatting the software demand-supply gap can come only if we learn to manage a large number of variables skillfully and if the components to the overall solution integrate well. Completeness and integration are, therefore, two key concepts in our vision. [Boehm, Standish, 1983, p. 30].

Front-end planning with early test plan scenarios integrates each of the forms of development methodologies into a single, powerful approach. Rigorous control of information is essential for unambiguous specifications. Structured analysis, such as SADT, requires that the definition of an information entity is consistent and correctly categorized within the overall informational hierarchy. Terms are explicitly and directly linked with their representations within the system's data dictionary, glossary, and its structural diagrams [Marca and McGowan, 1988, p. 135]. CASE tools such as **T** and protoyping can enhance front-end planning with early test plan scenarios. The approach of this thesis is a further refinement of the Life Cycle method. Front-end planning with early test plan scenarios does *not replace* any other methodology--it *improves* them.

CHAPTER 6

CONTINUING TO RESEARCH EARLY TEST PLAN SCENARIOS

The most incomprehensible thing about the universe is that it is comprehensible. Einstein

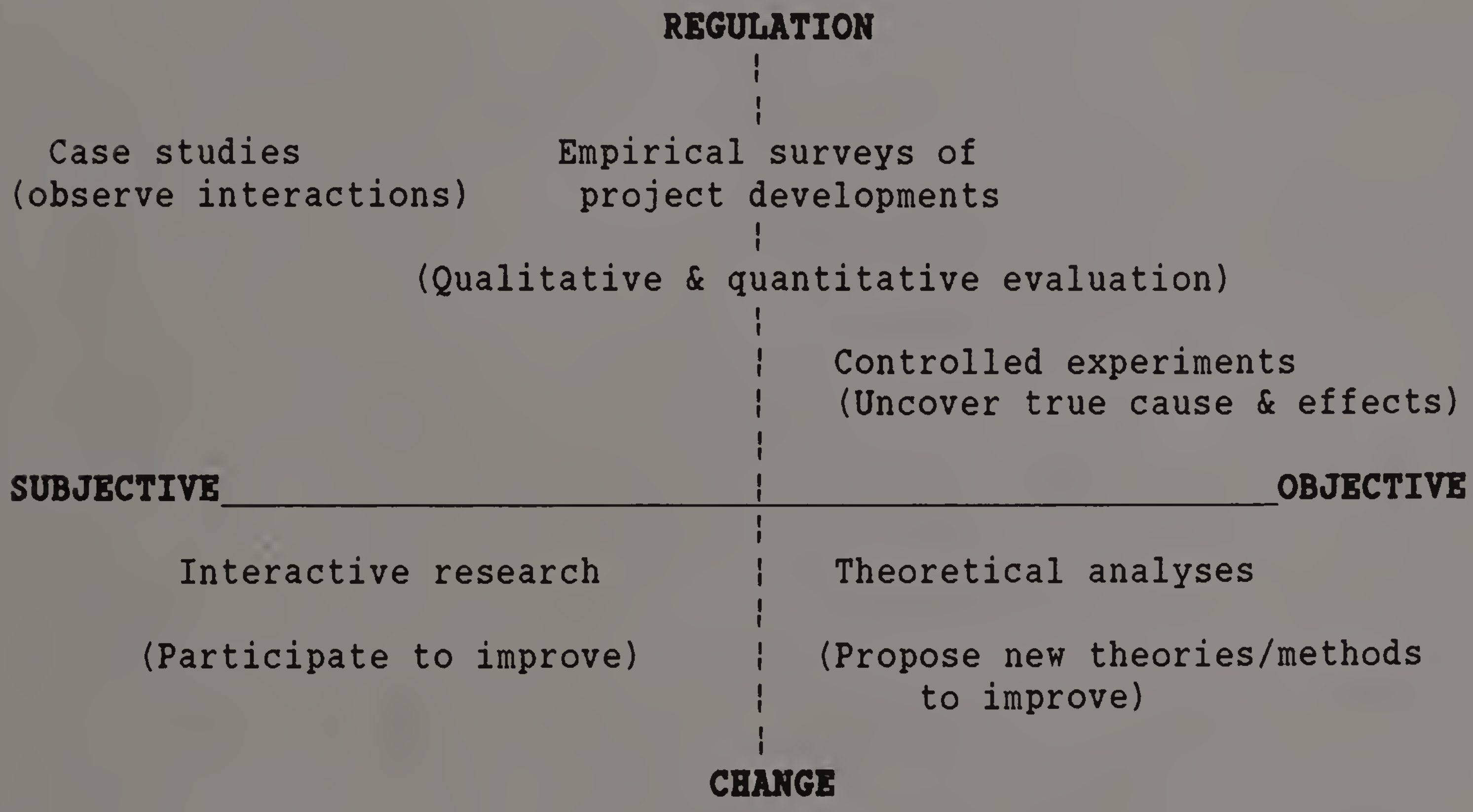
6.1 Research to Evaluate Systems Development Methodologies

As computer technology continues to accelerate its power and possibilities, there is a struggle to develop and evaluate methodologies to harness this potential into useful systems for organizations. While many methodolgies claim to offer great productivity benefits, software developments themselves continue to fail to meet anticipated time, cost and useability goals.

Researchers use historical development statistics to validate their findings with "experimental data and other empirically derived guidelines" [Daly, 1977, p. 235]. Uncertainty characterizes research on systems development. The young field does not universally accept particular research methods as valid. Systems professionals who are actively building systems are interested in case histories of successes or failures, viewing them as a source of insights. People who build systems are often wary of "proven" rules, preferring principles which help them accomplish their activities. "The essential thing is that we continue confronting our theories with new data and that we not be afraid to modify theories in light of such confrontations" [Attewell and Rule, 1984, p. 1191].

Five major research methodologies are currently being used to evaluate development methodologies: (1) Case studies of project developments; (2) Empirical surveys of developments; (3) Controlled experiments; (4) Theoretical analyses; (5) Interactive research.

There are differences between the purpose and the impact of these research approaches. Burrell and Morgan [1985] identified parameters to categorize research methodologies (see Figure 6.1).


REGULATION
Case studies (observe interactions)
Empirical surveys of project developments
(Qualitative & quantitative evaluation)
Controlled experiments (Uncover true cause & effects)
SUBJECTIVE
OBJECTIVE
Interactive research
(Participate to improve)
Theoretical analyses
(Propose new theories/methods to improve)
CHANGE

Based on Burrell & Morgan's four sociological paradigms.

[Burrell and Morgan, 1985, p.29]

Figure 6.1 Research Methods within Burrell & Morgan's Sectors.

Qualitative research highlights subjective, non-quantifiable factors while quantitative research is limited to measureable, objective data. Regulative research argues that there are solid, unchangeable laws which regulate behavior, with the purpose of research being to improve a situation by discovering these cause-effect laws.

Interactive research argues that human volition can create changes, that human activity is not restricted by underlying sets of laws. The purpose of interactive research is to bring about change, to improve a situation through proactive research.

6.2 Case Study Research

Case study research involves intensive studies of a single case or comparative studies of two or more cases. Case studies, although representing only a small portion of published research on systems development, are acknowledged to be rich sources of hypotheses. Case studies of individual cases were the foundation of organizational behavior. "One wonders what the impact on the field would be if another period of such case studies occurred [Lundberg, C., 1984, p. 39]. Lucas, one of the founders of research on the development of information system, recognized in the early 1970's that organizational behavior was a critical element in the development of successful information systems.

> It is our contention that the major reason most information systems have failed is that we have ignored organizational behavior problems in the design and operation of computer-based information systems [1975, p. 6].

While many case studies focus on the adaptation of users to new information systems, few examine the development process. If case studies are particularly effective in developing new insights in situations which involve the behavior of individuals in the organizational situation, and if the development of successful information systems is accomplished by individuals in an organizational situation, then case studies are a source of valuable information on improving the systems development process.

Poston and Bruen published a case study which examined the use of the **T** automated test generator during a systems development project at Leeds & Northrup Systems. This case describes current problems associated with development errors, and the improvements in quality and productivity which resulted from the use of the **T** software generator prior to coding. The case describes the organizational factors, the technical factors and the economic factors which influenced the development. Leeds & Northrup's state of practices before the use of the **T** software was compared to the state of their practices with the use of **T**. Details of the project plans and actual schedules were presented. When industry-reported values were available, quantitative measures of project size, failures, time and cost were compared with industry-average values.

The Leeds & Northrup Systems case study included five techniques for improving the development process. Before the case study, the researchers and practitioners decided that there was no need for separate studies to determine the contribution of each technique as each one had a history of commercial successes. The important question for Leeds & Northrup, and for the researchers, was the total impact of all techniques executed together [Poston and Bruen, 1987].

A case study was the foundation of the front-end planning early test plan scenario approach [Eastman, 1986]. A complex systems development was failing after three years of effort. When the approach described in this thesis was integrated into the development process, a dramatic turn-around in the development process occurred.

Organizations such as the US Army and AT&T use front end planning and early test plans. Senior design engineers at Digital Equipment Corporation, trying to improve the practice of systems development, are use early test plans. There is opportunity for additional case studies to increase understanding of systems development.

Case studies may be both qualitative and quantitative [Swanson and Beath, 1988]. While many case studies emphasize qualitative insights, some case studies describe mostly quantitative results. If case studies of complex developments were produced more frequently, there would be a wealth of information available for comparative studies.

Swanson and Beath [1988] used case study data in a controlled research situation in which they compared several developments. They used quantitative data gathered in standard questionnaires in order to compare the results of the studies. While Swanson and Beath's approach offers standardized information to compare the impact of different development methodologies, the structure of their research design limited the issues explored. By using standard questionnaires, and specified people being questioned, the case studies may not identify the significant issues which are changing the situation.

In Swanson and Beath's study, only systems *developers* were questioned. In the development situation, the sponsors and users play a significant role. The developers' opinions, although relevant, may obscure significant problems which are obvious to sponsors or users.

6.3 Empirical Research

Empirical research applies quantitative methods to observe, measure and define things and events of the "real" world. Several forms of empirical research could contribute to research: survey research, scientific research, triangulation, and controlled experiments.

6.3.1 Survey Research

Some empirical research does not attempt to develop or the prove theories, but only to understand the current condition of project development. These studies are necessary to ensure that the academic world of systems research is not isolated from the world of systems practice. Necco, Gordon and Tsai [1987] surveyed ninety-seven organizations to determine which technologies they were currently using, of these, which they found useful, and of those which they were not currently using, which techniques they were planning on using. Necco, Gordon and Tsai's research revealed a systems development environment in which a variety of tools are currently being used. Structured approaches, prototyping, and information center approaches were not seen as separate techniques which replaced traditional methods, but rather as tools which complemented the traditional Life Cycle approach.

Survey research is appropriate for methodologies (such as the one presented in this thesis) which employ a variety of techniques, performed in a sequential order. Of particular interest would be the current position of operational test plan development in projects. There are indications that many corporate developments are moving test plan development to earlier stages of the development cycle. A survey could be used to confirm this observation.

Front-end planning with early test plan scenarios is a flexible approach that can be improved through practice and research. The sequence of activities and the combination of approaches are critical factors in the front-end planning early test plan scenario approach which were not discussed in Necco, Gordon and Tsai's research. Survey research is be a powerful tool to understand not only what tools system developers are using, but also the sequence of these tools and the methods used to combine these approaches.

6.3.2 Scientific Research

Some theorists researching systems development assume that a careful compilation of data will reveal fundamental variables, which have stable causal relationships. They assume that the process of systems development can be understood through scientific methods.

These theorists consider the current status of systems development to be faulty because of a lack of understanding of the fundamental development variables. These theorists assume that "scientifically" derived understandings will reveal underlying relationships.

> The more clearly we as software engineers can understand the quantitative and economic aspects of our decision situations, the more quickly we can progress from a pure seat-of-the-pants approach on software decisions to a more rational approach which puts all of the human and economic decision variables into clear perspective. Once these decision situations are more clearly illuminated, we can then study them in more detail to address the deeper challenge: achieving a quantitative understanding of how people work together in the software engineering process [Boehm, 1984a, p. 20].

Development projects normally require documentation of change requests to correct code or design problems. Basili and Perricone [1984] collected all of these change requests on a medium sized project to analyze the types and quantities of errors which occur at different times during a project. The method they used did not interfere with the project itself--these change forms were a normal part of the development process.

Basili and Perricone, through detailed analysis of the change data which represented errors in requirement specification as well as design and coding, found that modifying an existing module resulted in an increase in errors of commission while creating new modules resulted in an increase in errors of omission. Surprisingly, errors in large modules were less frequent than errors in small modules [Basili and Perricone, 1984, p. 49].

The error detection methodology used by Basili and Perricone is an effective tool to understand the impact of systems development techniques, when the quantitative analysis is superimposed on a qualitative case study. After the research for this thesis was completed, an error analysis was completed using Basili and Perricone's technique.

This error analysis confirmed that the most costly errors were based in incomplete specifications. *The most significant finding of this error analysis was that major specification errors were detected during the development and review of the early test plan scenarios* [Eastman, 1986].

This use of error analysis, performed on data from existing case studies, is a valuable tool in increasing our understanding of the most effective techniques, and sequence of techniques, for counteracting the impact of inaccurate, ambiguous and incomplete specifications.

6.3.3 <u>Triangulation</u>

While academic journals tend to support research which is either qualitative or quantitative, some researchers support triangulation, a combination of methodologies used in research [Jick, 1988, p. 364].

Most of the studies which use quantitative research methods examine systems designers, systems developers and their tools. The role of the user and of the sponsor in systems development research is largely avoided. While researchers are gathering more information on the isolated work of systems developers, they are neglecting the most costly problem of all--producing systems which not only work well, but are also useful.

Card, McGarry and Page [1987] studied teams of engineers working on twenty-two projects. They found that the use of technologies such as structured code, design schedules and documentation, increased the reliability of the software but had no impact on the productivity of the development team.

Alavi [1984] used questionnaires and interviews to obtain qualitative and quantitative information from project managers and systems analysts participating in twelve development projects in six different organizations. Alavi selected a group of small to medium sized projects for financial, management and transportation management functions. Although the questions focused on whether the users were enthusiastic, understood the prototypes, and shared the reference points of the developers, no users participated in this research. Alavi was really asking developers for their *impressions* of user involvement, not obtaining information from the users themselves.

In a systems development situation where the major problem is the failure to produce systems which are useable in the actual operational environment, users are the experts. Research which ignores users, and does not prioritize the direct questioning of users, may be failing to address the major problem confronting systems developers.

Norcio and Chmura [1988] monitored, evaluated and compared software development techniques used on two major projects. Their study required engineers to submit detailed, weekly reports on their progress. Because this submission was not a normal activity for these engineers, the act of collecting the data itself would influence the project and bias the research. By forcing engineers to consider the type of work they were accomplishing, to take time to consider and record their work, and by increasing their awareness that their work was being closely monitored, the researchers biased their results by forcing engineers to change their work process.

Norcio and Chmura offer extensive information on hours devoted to development tasks, and a variety of information on progress ratios. Their case study attempted to demonstrate the effectiveness of software development techniques during a software cost reduction project and several projects which involved software technology evaluation [Norcio and Chmura, 1988, p. 165]. Norcio and Chmura focus on quantitative measurments, particularly a progress indicator ratio (PIR) to indicate when a design phase is complete. Norcio and Chmura's PIR compares the accumulated amount of time spent discussing a system's design with the amount of time spent creating a system's design. When this PIR ratio becomes constant, they contend that the design is complete.

Does this ratio reflect the "gut feeling" that many systems developers have when they determine that the design is sufficiently complete to proceed to the coding stage? If so, there remain problems with the ratio.

"Completeness" contributes to, but is not the sole variable, in "goodness" in system specification. With early test plan scenarios, "goodness" is measured by the measurability, completeness and useability of the system's design. Neither the sophistication of the design nor the evaluation of the designer is sufficient to determine the adequacy of the design. Users and sponsors are needed to evaluate the impact of the design on the useability of the system.

Care should be taken when performing research which attempts to measure progress or success with different development methodologies. Different methods have different basic assumptions concerning what variables influence "goodness". Measures which prove successful while researching some developments may be unsatisfactory when researching developments in different environments using different techniques.

If the process of filling out forms does not actually contribute to successful developments, then an alternative method of obtaining quantitative data would be to videotape some of the systems development activity. Using the videotapes, impartial scorers could identify the type of activity and the amount of time spent by each participant.

The quantitative evaluation of videotaped activity would be particularly useful in determining which activities could be automated, and which forms of automation would be most useful. This form of research could help determine the most prudent forms of automation, focusing on enhancing, rather than replacing, human judgment.

6.3.4 Controlled Experiments

Laboratory settings may be used for controlled experiments which examine sub-components of complex systems developments. The process of developing a complex system is too massive, expensive and lengthy to be an appropriate subject for laboratory experiments. Much of the laboratory research on programming uses students for subjects. Students with a few years of classroom programming are the "experts" and students with little exposure to computers are the "novices."

While many of these research studies claim to be evaluating programmers, a better label would be "Empirical Studies of Student Programming" [Curtis, 1986, p. 257].

Many controlled experiments do not address the macro issues associated with complex systems, concentrating on issues of cognition, not on project development. Researchers conducting laboratory experiments may themselves be familiar only with individual programming problems, and novices in the complex field of systems development. Too often, researchers assume that their findings are transportable to real-life systems developments.

> Current research investigates how people express solutions, but not how they learn what the real problem was to begin with. In fact, one of the greatest problems on large projects is that customers do not state their full requirements, because they are unable to determine them. Research on helping people elaborate their full desires (i.e. more fully defining the problem structure) may have dramatic payoff to real programming environments [Curtis, 1986, p. 258].

Laboratory experiments could be useful in refining early test plan scenarios. There could be controlled experiments which focus on determining what techniques are most useful in increasing productive communication and innovation. Particularly amenable to laboratory testing would be techniques for simulation and for graphic displays of relationships between structures. The laboratory is not the place to determine if simulation, test plans, or structured analysis is useful during project developments. But the laboratory is a reasonable place to develop better techniques for manually and technically accomplishing the tasks needed to facilitate communication and innovation.

Mantha [1987] conducted a study of twenty professional systems analysts to evaluate the impact of two methods of modeling a system design from a case study of a user situation. Ten of the analysts used a data flow model and ten used a data structure model to design a system. These participants were mostly MBA graduates who had five to ten years of systems development experience. While the data structure analysts developed more complete designs, an understanding of the flow of data was essential for an adequate representation of the required system. The importance of this form of research is that it offers insights into the real problems faced by real systems developers.

Particularly relevant for research on front-end planning is the importance of both data flow and data structure modeling. The laboratory is an appropriate setting to develop the most effective sequence of these data manipulation activities. Controlled experiments may assist in understanding which areas of this process could, should and should not be automated. The laboratory setting could be a fruitful location for developing effective teaching methodologies to transfer data building skills to programmers.

> Why does structural modeling seem to be more difficult to teach and learn? What are the best strategies for training people in using these approaches? What are the typical problems they will encounter [Mantha, 1987, 542]?

6.4 Theoretical Research

Theoretical analysis proposes answers to questions raised by empirical experiences and research. The proposed methods and theories attempt to change and improve existing practices. Research on systems development is particularly volatile because technology is constantly changing. While research attempts to identify what is occurring, the situation itself changes.

> The frantic pursuit of repeatability and statistically significant correlations is based upon the belief that science is a search for laws. This has blinded many scientists to the need for careful description and analysis of what *can* occur and for the explanation of its possibility. Instead they try to find out what always occurs...and usually fail [Sloman, 1976, p. 17].

Many people who have been engaged in the practice or observation of systems development, analyze their experiences and the experiences of others. These people propose new methods and theories to address problems which were identified by empirical research. Theoretical analysis, originally based in empirical work, has progressed to the exploratory stage of innovation.

Boyd and Pizzarello [1975], supported by an Air Force contract, developed the WELLMADE^{tm} Design Methodology as a mathematically founded design discipline which could improve software development. Boyd and Pizzarello emphasize requirement specifications which are sufficiently complete to develop abstract program designs. "These specifications are obtained by constructing the program state space and the necessary assertions to define the input and output subspaces for all the required operations" [Boyd and Pizzarello, 1978, p. 277].

Barlow [1981] emphasizes the importance of project management on the success of a system development. Barlow produced a check list of events which must be satisfactorily managed by the successful project manager (see Table 6.1) prior to the actual development of a system.

Table 6.1 Barlow's Project Management Steps.

Step 1:	Define project objectives and goals.
Step 2:	Appoint the project manager.
Step 3:	Define scope of work & assign responsibilities.
Step 4:	Establish project organization and interfaces.
Step 5:	Define cooperation required.
Step 6:	Establish & document major work packages in project.

[Barlow, 1981, pp. 57-59].

The difference in perspective between Barlow, Boyd and Pizzarello exemplifies the complexity of the systems development process, and the importance of considering organizational, managerial, technical and logical perspectives when evaluating this process. Awareness of the diverse variables associated with successful systems development prompted Boehm to stress completeness and integration in his answer to the question, "How (can we) achieve a state-of-practice in the 1990's where we can build embedded computer system software with adequate levels of productivity, adaptability and reliability" [Boehm, 1983, p. 30]. Boehm considers economic, technical and organizational factors. Procurement policies will become more important as packaged software and third party developers become more commonly used. Boehm's analysis finds that the most critical need for the improvement of software development is improvement of its *process* rather than its *products*.

Ruskin and Estes [1986] analyze the helpful and harmful effects of organizational factors in systems development, proposing actions which could improve the development process by enhancing the positive and reducing the negative impact of these organizational factors. Ruskin and Estes, finding that the organizational environment determines the success or failure of a project development, theorize that project managers can "enhance their overall chance of success by understanding how organizations affect projects" [Ruskin and Estes, 1986, p. 4]. They list factors, helpful effects, adverse effects and possible counteractions for adverse effects of seven organizational factors (see Table 6.2).

Table 6.2 Factors Affecting Project Managers and Project Success.

(1) Organizational structure
(2) Staffing
(3) Client relations
(4) Attitudes toward risk
(5) Communications between users, sponsors and developers
(6) Expectations of all participants (economic, project usefulness, personal performance, personal reward, hours worked, satisfaction)

[Ruskin and Estes, 1986, p. 4]

Taking an opposing viewpoint on the development process, Blum [1987] asks how full automation could improve the systems development process by eliminating human error and imperfect work. Blums's analysis of the system process virtually eliminates all organizational considerations and all possibilities of human innovation, individual or social, during systems development. Blum's analysis produces a theory which envisions a development environment where the production of a system is totally divorced from the process of defining system requirements. Blum foresees a rise in the importance of the software factory. The software factory concept, widely used in Japan by firms such as Hitachi, promotes team work among systems developers. Any member of a team of thousands of developers should be able to satisfactorily develop a system, using an integrated set of automated tools.

Shemer, [1987] reviewing the problems associated with information systems development seeks the appropriate model for systems analysis. While many methods and processes focus on individual elements of the process, Shemer theorizes that developers should have a broader awareness of the severity of systems development process failures.

Shemer theorizes developments could be improved if the specification process included an analysis of organizational goals, components of the system, environmental constraints, resources, inputs and outputs and interrelationships between the components [Shemer, 1987, p. 509].

Theoretical analysis is not separate from, but complementary to, other forms of research. Basili and Rombach [1988] ask how we can develop sofware engineering processes which "integrate sound planning and analysis into the construction process" [Basili and Rombach, 1988, p. 770]. Basili and Rombach developed a model which contains principles which they "learned during a dozen years of analyzing software engineering processes and products" [Basili and Rombach, 1988, p. 770].

Their model stresses an awareness of the current status of a project, integrating planning for improvement into the project plans, and accomplishing project plans. A fundamental characteristic of front end planning with early test plan scenarios is the continued emphasis on the process of reviewing which *improves*, rather than simply *approves*.

6.5 Interactive Research

Software development researchers are often participant observers who actively participate in the activities which they are observing. They develop principles, derive implications from their principles, and become actively involved in the development process.

Participant observers address the same problems faced by empiricist researchers--unclear specification of critical system attributes, lack of appropriate engineering to achieve these attributes, and lack of appreciation for the dynamic and interactive nature of systems development [Gilb, 1988, p. 163]. Even empiricist researcher's principles, analyses and insights are "substantiated" by the author's statement of size and number of projects with which he/she worked. Daly's sets of principles were derived from "2,000,000 hours of software development experience" [Daly, 1976, p. 229].

In Europe, participative research is more prevalent than in the United States. In Norway, Max Elden performed research which involved empowering worker participation during the systems development process (1986). Elden's research involved observing, training and encouraging the active change of the roles of the workers. Norway has federal laws stating that employees have a right to approve any automation plan which will affect their jobs *before that automation plan is installed.* Workers have the legal right to participate in the process of designing the automated systems which will impact their work. Elden received funding to research and implement training processes to build worker skills to the level required for viable participation.

The current role of the users and sponsors is in many American corporations is reactive. Once a system is running they react, identifying problems and suggesting improvements. The front-end planning early test plan scenario approach methodology is successful only when users and sponsors assume a proactive role in which they identify problems and suggest improvements during the initial, planning phases of systems development.

CHAPTER 7

EARLY TEST PLAN SCENARIOS IMPROVE THE SYSTEMS DEVELOPMENT CONTRACT

It would be possible to describe everything scientifically, but it would make no sense; it would be without meaning, as if you described a Beethoven symphony as a variation of wave pressure.
Einstein.

7.1 Problems with Current Software Development Contracts

Tens of billions of dollars are spent by corporate America each year for the development of computer systems which are "typically 100% over budget and a year behind schedule" [Carroll, 1988, p. 1]. General Motors Corporation spends $3 billion and ITT Corporation spends $1 billion each year on software systems. Most complex systems are developed, under contract, by third party vendors. The massive number of dollars spent annually on software development, coupled with a high probability of major cost escalations, significant delays, and failed systems, have "spawned a proliferation of lawsuits rivaling the growth of the computer industry itself" [Mislow, 1985, p. 124].

7.2 Assumptions of Traditional Development Contracts

Courts struggle reactively with contract disputes in the form of lawsuits brought by disastisfied, often financially ruined, customers of software systems. Writers in the field of legal literature are trying to improve existing software contracts that support traditional assumptions of software development (see Table 7.1).

Table 7.1 Traditional Assumptions of Software Development.

(1) A software development contract describes the provisions of an exhange of money for a specific product.

(2) The software vendor has sole responsibility for the quality and timely delivery of the product.

(3) A more accurate and complete description of *what* should be delivered is the primary method of improving software development contracts.

The first assumption, that a contract describes a product, is supported by those who argue for an increase in warranty accountability for software vendors. Under the Uniform Commercial Code (UCC), vendors of goods can be held liable for their breach of either express or implied warranties, or both [Mislow, 1988, p. 125]. Vendors of services, where no tangible goods are produced, are considered to be legally responsible under rules of professional liability, not under the warranty provisions of the UCC [Freed, 1977, p. 281]. Legal debate continues concerning whether different forms of software development contracts should be considered the sale of services or the sale of goods. Although software development may be partially a service, and not fall under UCC regulations, some argue that all software should be subject to warranties. Mislow [1985, p. 124] is particularly concerned about the precendents that the courts are setting by absolving vendors of warranty accountability.

The second assumption, that software development vendors should be personally liable for the software they create, is receiving increasing attention [Chretien-Dar, 1987, p. 499]. This approach assumes that the software developer is similar to a skilled surgeon operating on an unconscious patient.

The third assumption is that the functionality of the planned system can be completely described before it is designed. This assumption is supported by those who attempt to solve the development crisis by increasing the accuracy and completeness of the description of the end product [Werstermeir and Millstein, 1988]. Those who argue that increasing the detail of the *What* of the contract will improve the development, deny that *How* a system is developed changes *What* is developed (see Figure 7.1).

The early test plan scenario approach challenges these three basic assumptions by asserting that the software development contract should be a promise to collaborate to develop a structure to modify an organization's current operational environment. When organizations contract with software developers, they promise to participate in a temporary, collaborative partnership. The development contract establishes the characteristics of this collaborative partnership and the process by which sponsors, users and vendors will work together to produce the system.

Early Test Plan Scenario Contract

1. *How* a system is developed affects *What* is developed.

```
 ______________________________
|                              |
|             WHY              |
|_:_:______:_:__________:_:____|
  : :      : :          : :
  : :      : :          : :
  : :     _\/__        _\/__
  : :    |     |      |     |
  : :    | HOW |<->   | WHO |
  : :    |_:_:_|      |_:_:_|
  : :      : :          : :
  : :      : :          : :
  : :      : :          : :
  : :      : :          : :
  : :      : :          : :
 _\/_______\/___________\/_____
|                              |
|            WHAT              |
|______________________________|
```

2. The successful development of complex systems requires the collaboration of sponsors, users and developers throughout the entire development process.

3. The purpose of the contract is to establish the boundaries, rules and controls of the collaborative process.

Traditional Contract

1. *How* should support but never change the *What*.

```
 _______________
|               |
|      WHY      |
|______:_:______|
       : :
 ______\/_______
|               |
|     WHAT      |
|______:_:______|
 ______\/_______
|               |
|      WHO      |
|______:_:______|
 ______\/_______
|               |
|      HOW      |
|_______________|
```

2. The sponsors determine *Why*, sponsors and users determine *WHAT*, developers alone determine *HOW*.

3. The purpose of the contract is to define the characteristics, costs and deadlines of the final product.

Figure 7.1 Comparing Early Test Plan Scenario Contracts to Traditional Contracts.

7.3 Provisions of an Early Test Plan Scenario Development Contract

A software development contract, influenced by the early test plan scenario approach, would establish the boundaries and structure for the entire development process, including a sequence of activities and criteria for measuring the successful completion of each sequence. The contract should include explicit agreements concerning the management of the systems development process, including the procedures for resolving conflicts and for establishing the specific conditions of each sequence.

The contract would establish the mutual responsibilities of the collaborators, with provisions for monitoring these provisions. The exact balance of accountability would be different, and separately specified, for each sequential phase of the development process.

The main development contract establishes the development phases, and guidelines for producing each subcontract. At least eight sub-contracts are needed for a development (see Table 7.2). Different vendors may be selected for each subcontract. Products of prior phases support activities of subsequent phases. The main development contract could be with a vendor who is not the vendor for any subcontracts, similar to a building contractor who arranges all subcontracts.

The early test plan scenario contract is more rigorous, has more subcontracts, and requires more specific allocations of responsibilities among users, sponsors and developers than does the traditional contract. The initial cost of the early test plan scenario contract would be higher than the initial cost of the traditional systems contract.

The economic benefits of the early test plan scenario contract, should be measured by the costs required for the timely, successful completion of the entire project. The economic benefits of early test plan scenarios are based in lower total costs for systems that become useful within organizations.

Table 7.2 Early Test Plan Scenario Contracts for Developments.

Early Test Plan Scenario Main Contract

A. Establish the characteristics of the development process.

B. Establish the rules of the collaborative process.

C. Establish the methods of forming each subcontract.

D. Establish the monitoring and control mechanisms.

E. Establish the process of permitting change.

Early Test Plan Scenario Subcontracts

1. Definition of objectives.
2. Description of functional needs.
3. Analysis and design of system structure.
4. Test plan development and review.
5. Coding of software.
6. Testing of software.
7. Fit system into operational environment.
8. Modify (maintain) system.

7.4 Contracting for Goods or for Services?

Much legal attention is focused on determining whether systems development contracts are for services or goods. The legal system provides more stringent vendor accountability for goods, through the UCC, (Uniform Commercial Code). Under the provisions of the UCC, the vendor of goods is held liable for both express and implied warranties of the goods that he/she sells. The courts define goods as "moveable things." Contracts in which developers actually deliver programs on disks or tape to a customer are considered to be for "goods" while those in which developers work directly on the customer's computers are considered to be for "services."

Confounding the issue are current powerful communication and network technologies. A vendor in California can spend months developing a system directly on the Massachusetts customer's computer. A few years ago, the vendors, performing exactly the same work, would work on their own computers and deliver tapes that contained the finished software to the customer. Technology continues to outdistance the legal system's interpretations.

Whether the development contract is considered by the courts to be for "services" or for "goods" has a significant impact on the outcome of disputes. "Goods" are covered under the UCC. "Services" are governed by the laws of general contract law. If software is "goods", and falls under the UCC, it automatically has implied warranties, which are in effect unless expressly and carefully disclaimed. "U.C.C. article 2-314 (1982) supplies the warranty of `merchantability that arises when the seller is a merchant as to goods of that kind and requires that goods be merchantable'" [Chretien-Dar, 1987, p. 478]. To be "merchantable", goods should "(a) pass without objection in the trade, and (b) in the case of fungible (exchangeable) goods, be of fair average quality,...(c) be fit for the ordinary purpose for which such goods are used" [Chretien-Dar, 1987, p. 479]. Services, governed by contract law, have no such implied warranties.

The UCC code transforms most vendor statements into express warranties, with the customer not having to prove that the vendor intended her/his statements to be interpreted as warranties. These express warranties are applicable only to development contracts that the court rules are for "goods", not for contracts for "services."

Express warranties are created by: "any affirmation of fact or promise relating to the goods, (2) any description of the goods, or (3) any sample or model" [Chretien-Dar, 1987, p. 479]. While the courts do not allow software vendors to disclaim all express warranties in a blanket statement, they do allow specific disclaimers which have the effect of blanket disclaimers. In *APLications, Inc. v. Hewlett-Packard Co.,* both implied and express warranties were considered by the courts to be sufficiently disclaimed by the words: "NO OTHER WARRANTY IS EXPRESSED OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTY OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE" [Chretien-Dar, 1987, p. 490].

Contracts which support early test plan scenarios would include a main contract for the process of systems development and subcontracts for services and goods. These contracts focus on the overall process of systems development. Certain subcontracts are for goods, specifically the subcontract which is for the actual code. Other subcontracts are very clearly for services. One conflict inherent within the traditional contract, and its interpretation under the UCC, is that any contract which involves a good, is considered to be, in its entirety, a contract for a good.

The end product, the software code, is simply the final step in a complex process. Most systems developers consider the actual coding of the system to be the simplest activity of the entire process. When the preceding phases have been skillfully and thoroughly accomplished, a development contract could actually omit the "final product", leaving the actual coding activity to the customers.

The attempts by many legal researchers to improve the traditional software development contract improves the subcontract for the actual coding of the designed system. During the coding stage, the primary responsibility is for programmers to finalize the system by adding the technical detail to the overall structure and design. The programmers, during coding, are following directions and firm guidelines in the concretization of a system. Programmers can be held accountable for the reliability of their code - but not for the design and impact and useability of the system itself.

7.5 Legal Views on Systems Development

Articles on contract law support the viewpoint that expert software vendors develop and deliver something, either a good or a service, to a user, in exchange for a predetermined sum of money. There are assumptions in this approach which can directly increase customer-vendor conflict and decrease the potential for successful systems developments. This approach, which views the final, coded system as the primary object of the contract, contributes to the tremendous surge of legal conflicts resulting from failed development contracts.

7.5.1 Naive Users and Unconscionability

The vision of a naive user buying either a good or a service from an expert developer perpetuates the notion that software development is simply a sales transaction between two independent parties. One offers money and "blueprints" [Chu, 1982] for a product, and the other produces the product which fits these "blueprints". The user is urged to make the best "bargain" while the vendor focuses on acquiring the greatest reward with the least effort. The denial of essential collaboration becomes an integral part of the traditional contract, and reduces the opportunity for a successful development process.

Customers are naive when they believe that the expenditure of funds to a skilled software developer will solve an organization's problems. Customers are naive when they choose not to recognize the critical importance of their expertise and contribution to the system development process.

The doctrine of unconscionability, a provision under the UCC, gives courts the authority to void some or all of a contract, if it finds that the terms of a contract are "unreasonably favorable" [Levin, 1986, p. 2108] to one party. In the past, the courts have reserved this doctrine for "disadvantaged consumers, typically in installment contracts" [Levin, 1986, p. 2108]. Courts do not find unconscionabilty in most business situations, considering businesspeople to be sophisticated and aware when they sign contracts.

Recently courts have become more liberal in their application of unconscionability to software contracts [Levin, 1986, p. 2109]. When courts apply the doctrine of unconscionability, they often find that the purchaser of software did not know enough to make a valid choice, and therefore the terms of the contract were unfair to the customer.

Some legal researchers argue that unconscionability could be found "in commercial transactions for computer products where the vendee lacks the sophistication necessary to create bargaining equality" [Levin, 1986, 2110]. When system development is viewed as a collaboration, requiring effort and commitment on the part of both customer and vendor, the issue of unconscionability is fundamentally different. The obligations and responsibilities of both customers and vendors are to increase each other's understanding and participation, to continually evaluate and improve, to work together to create the best possible system.

A limited view of systems development, which freely applies unconscionability to a contract situation, may be compared to a medical situation. An overweight person asks a doctor for a diet pill, and pays the doctor a fee. Assuming that the pill will cure his problem, the overweight person reduces his exercise and eats even more. When the person gains weight, he sues the doctor because the pills did not work. When software is viewed as a "diet pill" which will work if the customer's needs are sufficiently described, if the customer pays a sufficient amount, and if the vendor is sufficiently skilled, the crucial role of customer participation is ignored.

Recent court decisions which promote unconscionability and stress consumer rights and satisfaction obscure consumer responsibility. These decisions relieve customers of an obligation to become involved in the design and development of their systems. Systems development contracts fail when they ignore the collaborative nature of complex systems development. Court remedies to address systems development contract failures should not impede or diminish the responsibility of the customers to collaborate with vendors during the development of systems which will impact the customers' organizations.

Levin argues that the relative bargaining powers of the parties involved in a contract must be assessed to determine if a court ruling should alter the terms of a contract. In complex developments, both customers and vendors are both astute businesspersons. The argument that a customer of a complex system is naive, is invalid. Decision makers in organizations which have sufficient resources to sponsor complex developments (and often employ lawyers) are *not* naive.

> There are three important tests to determine whether a corrective rule achieves the optimal balance between upholding freedom of contract and compensating for imperfect market conditions: (1) whether the scope of the proposed rule fits the scope of the existing problem; (2) whether the effective life of the rule will coincide with the duration of the problem; and (3) whether the rule reduces the incidence of problematic transactions by encouraging the free flow of information throughout the industry [Levin, 1986, p. 2120].

When courts find unconscionability in situations where there is disparity in technological sophistication between the parties, they may be limiting the potential for improving the system development situation [Levin, 1986, p. 2139]. Systems fail when they are not useful. While vendors may be the tecnical experts, customers remain the experts in useability.

Laws, such as the Molina Bill #1507 in California, attempt to require the inclusion of warranties in all sales and leases of computer products and in their advertised claims. The bill mandates that there can be no time limitation on warranties, and that a sponsor can revoke acceptance of a system at any time [Levin, 1986]. This bill, and other similar attempts at protecting customers, will deteriorate the system development process if it encourages customers to refrain from participation, assured of their ability to sue the vendor if they do not like the product.

7.5.2 Customer and Vendor--Interdependent or Collaborators?

The traditional systems development contract considers the customer and vendor to be interdependent on each other. Interdependency encourages each party to protect their own interests, to distrust the motives of the other party, and to attempt to shift the balance of rewards and costs to their own favor. Traditional development contracts which are founded in interdependency promote disputes and conflicts (see Table 7.3).

The development process of custom computer systems is burdened with recurring problems that create disputes between the user and vendor, and often lead to litigation. First, because customers generally know less about complex computer systems than do vendors, customers tend to play a passive role in the development process, relying on the skill of the vendor. This reliance hinders the development process. The vendor needs significant input from the user to design and develop a system which fulfills the user's needs.

Table 7.3 Impact of Limited Viewpoints on Software Development.

Limited Viewpoint	Overlooked Viewpoint	Impact of Limitation
1. Software development contracts are for goods or for services.	1. Software Development is a process which creates a system which changes an organization.	1. The usefulness of a system is ignored when its impact is being established.
2. A naive customer buys a complex product from an expert developer.	2. Expert customers share responsibility and work with expert system developers.	2. Law perpetuates the myth of "them" against "us".
3. Users define *what*, developers define *how*.	3. The *how* changes the *what*.	3. Users do not contribute to the system design.
4. When a development is complete, its value can be measured be determining how completely the final system satisfies the original functional requirements.	4. The value of a software development is measured by how useful the final system is to the organization.	4. When system is logically correct and operational, conflict develops as users, sponsors and developers have different interpretations of functionality.
5. The developer is the only expert.	5. Successful system developments requires the expertise of users, sponsors and developers.	5. Total development cost and time escalate greatly if sponsors and users do not evaluate a system's usefulness until after it is operational.
6. Critical contract question is "How should the proposed system work?"	6. Critical contract question is "How should the development process be structured, managed, and documented.	6. When the focus is on the final product instead of the interactive development process, the final system may disrupt the organization.

Even the technologically unsophisticated customer should play an active role in defining requirements, making design decisions that affect the system's cost or ease of use, and performing system testing (see Table 7.3).

The customer's failure to participate in the systems development process may promote customer-vendor disputes. The customer may develop unrealistic expectations about the performance of the system, or the vendor may misinterpret the customer's needs. The customer may not understand the final system, thus reducing its effectiveness and accuracy [Anderson, B., 1985, p. 1547].

7.5.3 Rolling Estoppel--Refuting the Right of the Vendor to Fail

"Rolling estoppel" is a concept which may be added into a software development contract, particularly one which will extend over a lengthy period of time. Rolling estoppel requires "a party to object at certain intervals if it is aware of a problem or delay, regardless of whether that party is at fault. A party that fails to object is estopped from complaining later about that problem or delay" [Gordon and Starr, 1985, p. 490]. Including rolling estoppel in a contract supports collaboration among vendors and customers. The contract can contain specific requirements for periodic project status meetings, during which all parties must present information on each of those delays and problems of which they are aware, regardless of the source of the problem or of the person responsible for the problem. Rolling estoppel in a computer contract forces any party who notices a problem to identify that problem, regardless of the source of the problem [Gordon and Starr, 1985].

If there is no clause in a development contract for rolling estoppel, the sponsor may deliberately refrain from identifying problems caused by a vendor, thereby guaranteeing the "right of a vendor to fail." With traditional contracts, it can benefit the customer to let the vendor fail. If the customer no longer wants the system, the customer can subtly but deliberately refrain from identifying misconceptions or other vendor errors. When these "errors" are contained in the final system, the customer can sue the vendor for creating an "unquality system." Political games thrive in contracted developments.

7.6 Contract as Promise to Collaborate

The front-end planning with early test plan scenario approach of this thesis considers the main contract of systems development to be a contract for innovative collaboration. The underlying premise of this approach is that customers and vendors promise to work together, to respect the expertise of each other and to devote their energies, skills and experiences to the collaborative effort (see Table 7.4.). The concept of "contract as promise" [Fried, 1981] fits this approach.

Table 7.4 Questions Asked by a Front-End Planning Contract.

(1) How will the customer-vendor relationship be structured to enable and enforce customer-vendor understanding, evaluation and improvement throughout the development process, particularly during the design of the software?

(2) What are the explicit responsibilities and contributions required of the customers and vendors during the development?

(3) What are the operational scenarios the system must satisfy? When will the system be "good enough" for the pre-operational phases to end?

(4) How will customer-vendor responsibilities change over the course of the development?

(5) What development structures and standards will be followed to facilitate post-operational modifications?

7.6.1 Contracts for Complex Developments

Complex developments are inherently different from simple programs. We have reached a stage, technologically, where we are able to create monster systems which we are unable to either understand or control.

Many legal remedies attempt to improve development contracts. These remedies are useful but insufficient. It is useful to improve contracts for the coding activity. It is inappropriate to apply the same concepts to the entire development. A common remedy for development failures is to improve the accuracy, completeness and clarity of functional specifications [Gordon and Starr, 1985, p. 493].

Improving specifications is necessary but insufficient. Interpretations of specifications will differ. Collaboration must occur throughout the development process to ensure that vendors and customers are sharing the same interpretations.

7.6.2 System Development Contracts as Promise among Experts

Law is based on underlying, unifying structures which are moral principles. The Promise Principle, the respect for self-imposed obligations, is fundamental to contract law [Fried, 1981]. The traditional contract's premise is that the customer promises to pay the vendor, and the vendor promises to fulfill the needs of the customer. The front-end planning with early test plan scenarios contract supplements these promises with additional, self-imposed obligations which support vendor-customer collaboration.

Collaboration implies equality in expertise. Traditional contracts try to help the "inexpert" customer. Mislow argues that new theories of legal liability are needed to "provide an effective remedy for naive users who purchase deficient computer systems" [Mislow, 1985, p. 129].

The approach used in this thesis rejects the concept of a "naive" user. The user is the expert in the organization's operations. The developer is the expert in computer technology. Together they collaborate to develop a technical system which improves the organization's operations (see Table 7.5).

7.6.3 Contracts for Correct, Useable and Useful Systems

The vendor in the traditional contract promises to create systems which correctly meet the user's functional requirements. In the front-end planning with early test plan scenarios contract, the vendor still promises to build a "correct" system.

Table 7.5. Comparing Vendor and Sponsor Contract Obligations.

Traditional Contract	**Front-End Planning with Early Test Plan Scenarios Contract**
Vendor Obligations:	*Vendor Obligations:*
1. Satisfy needs of customers by using own expertise to build a system which meets the needs of the customer.	1. Satisfy needs of customers by contributing own expertise to the collaborative effort focused on building a system which meets customers' needs.
	2. Ensure customers are able to evaluate their needs.
	3. Ensure customers are able to understand their options.
	4. Ensure customers are able to evaluate the cost and usefulness of each option.
	5. Fully share personal skills, expertise and experience to enhance customers' efforts.
	6.Ensure that customers are aware of costs and benefits of process, technical and standards options.
	7.Formally confirm agreements.
Customer Obligations:	*Customer Obligations:*
1. Pay the vendor.	1. Pay the vendor.
2. Give the vendor sufficient information to produce a system.	2. Actively collaborate with the vendor to uncover all information on organization necessary to design a system which will improve its operational environment.
	3. Require and permit active participation of all relevant employees.
	4. Understand the operational impact of all proposals.
	5. Evaluate the useability of system proposals.
	6. Improve the usefulness of system proposals.
	7. Formally confirm agreements.

In addition, the vendor and customer, both promise to collaborate to develop a system which is not only correct, but is also *useable and useful*. A useful system is often less elaborate, less complex, less automated - designed to improve rather than to disrupt an organization's operational environment.

When collaboration is an essential part of a contract, the quality and success of the vendor's effort is measured by the fruitfulness of customer's collaboration. Vendors are successful when customers participate fully in the development process, using the vendor's expertise to produce the best possible system to improve the operational environment of their organization.

7.6.4 Rules of Collaboration: "I Owe You" and "You Owe Me"

There are no widely accepted rules or conventions which govern collective innovation in our current social and economic environment. In the absence of generally accepted conventions, front-end planning with early test plan scenarios offers a rigorous set of procedures and rules to define and to enforce the obligations during the critical first stages of the collaborative effort. Including a set of procedures in systems development contracts offers a set of guidelines for promoting, fulfilling and evaluating the collaborative obligations.

The contract can formally state the "I Owe You" and "You Owe Me" obligations between customers and vendors. These mutual obligations provide the structure and boundaries needed to guide the type of innovative collaboration which produces systems which are correct, useable and useful within the organization.

7.6.5 Principles of Restitution

There are two principles which apply to restitution when contracts are considered willful promises. The victim of the breach is entitled to no more or less gain than would have occurred had there been no contract. The second principle is that restitution should cover restitution for harm which was caused by one party relying on the other party's promise [Fried, 1981, p. 17].

In a traditional development, the failure of a system to operate correctly is a condition for restitution, with little consideration for the causes of the system failure. Only if the customer withheld vital information from the vendor, might the vendor not be held legally responsible for the failure [Westermeier and Millstein, 1988, p. 85].

In a collaborative contract, restitution for the failure of the end product requires many considerations. The contract is for the entire development process. There is a great deal to be gained during the process as well as from the final product. Failure of the final product may instigate the rapid development of a much better product.

In 1982, a state agency, the SKILLS Center in Springfield, Massachusetts, contracted with a software vendor for the development of a computer system which could be used to match people who were being trained by the agency with organizations who had available jobs. After two years of work, the software vendor's system was inoperable. The analyst who had been developing the system left the employ of the vendor, leaving behind poorly documented software which had to be scrapped. When the SKILLS Center contracted with another software vendor, a single analyst produced a useful, operable system, which fully met all of the agency's requirements, in less than two months [Eastman, 1987].

The second vendor was not twelve times more skillful than the initial vendor--but the customer had two years of experience, working with the initial vendor, understanding the agency's needs and evaluating the agency's options.

The two years contributed significantly to the successful merging of a system into the agency's operations. The failure of the first system simply delayed operations by two months. The new system, designed from the beginning as a collaborative effort, benefitted from the expertise developed by the agency during their two years of work with the initial vendor.

The principle of restitution is dramatically altered by the collaboration principle. The approach offered in this thesis (and its explicit collaborative innovation), is not designed to protect either customer or vendor in the event of system failure. The approach is, instead, designed to prevent the failure itself. If either party reneges on the opportunity and the responsibility for encouraging active involvement and viable collaboration, the system development process itself suffers.

A fundamental function of an front-end planning with early test plan scenarios contract is the formal statement of obligations, of lists of "I Owes." These obligations are rules of collaboration which are formally included in a development contract to guide the development, to measure each participant's contribution, and to be the basis for evaluating restitution and culpability if the development flounders.

Front-end planning with early test plan scenarios improves traditional contracts by formalizing the relationship between the customers and vendors. The agreement of the two parties to work together is often labeled as "good will" or "care and concern." The approach of this thesis formalizes intent into measureable actions, reducing the ambiguity of traditional development contracts. The front-end planning with early test plan scenarios approach is a set of rules that structure the activities associated with the will of the parties.

> The language of mistake suggests certainty is the paradigm, but in fact contracts are largely a deliberate attempt to deal with uncertainty... The straight forward point is that two parties, though they seemed to have agreed, had not agreed in fact. And this - not mistakes or risks - is at the heart of these cases. There is just no agreement as to what is or turns out to be an important aspect of the arrangement...The one basis on which these cases cannot be resolved is on the basis of the agreement, that is of the contract as promise. The courts cannot enforce the will of the parties because there are no concordant wills. Judgment must therefore be based on principles external to the will of the parties" [Fried, 1981, pp. 59-60].

7.6.6 Vendor Malpractice

Many attempts to address the tremendous burden of systems development ignore the complexity, the difficulty and the extreme effort which must be committed to the development of complex systems. This commitment is not trivial, it requires a level of intensity and expertise which is not amenable to attempts to find easy solutions.

One "easy" solution which is becoming increasingly popular, is an attempt to hold the systems developer professionally liable for systems development failures. The systems developer would be subject to the same malpractice suits as the surgeon [Mislow, 1985, p. 129].

Professionals find this particular movement a serious threat to improving the development process. A doctor's skills and training is based on knowing how the human body functions and on understanding techniques for improving its functions. In successful software developments, there is collaboration between three groups: users, sponsors and developers.

When developments fail, customers lose not only the resources given to the vendors, but other resources spent as the organization prepared its people, machines and other systems for the new acquisition. When systems are delayed, there are additional costs for developing temporary operations to carry the organization until the system is available. Frequently, these temporary operations become fully integrated into the operational environment, totally eliminating the need for the delayed system. The complex system, finally operational after lengthy delays and massively escalated costs, is no longer needed and is scrapped.

Courts struggle with allocating financial responsibility for the billions of dollars spent on faulty developments [Westermeier and Millstein, 1988]. The legal profession has fascinating, new situations to explore. But the process of developing systems is not improving.

The front-end planning with early test plan scenarios approach requires active collaboration between vendors and customers, is not an easy way out of the current, very expensive turmoil. Participants in the development process have to be proactive, they have to plan, they have to listen, they have to evaluate, they have to improve. Both customers and vendors have to respect each other while actively contributing their expertise. Customers and vendors both have to work extremely hard, with collaborative innovation, to produce a successful system.

CHAPTER 8

IMPROVING SYSTEMS DEVELOPMENT IN THE 1990'S, AND BEYOND

Where we are going, we don't know. But we are going there.
Philippe Kahn.

People accomplish more when they work together than when they work apart. Management's task is "to make people capable of joint performance, to make their strengths effective and their weaknesses irrelevant" [Drucker, 1988, p. 75]. Cultural diversity and technological complexity will make working together in the 1990's even more critical and even more challenging. Individuals will need to assume personal responsibility, be creative, and collaborate with others [Reich, 1988].

Prior to the Industrial Revolution, craftspeople and tradespeople worked directly with their customers, providing customized services to meet unique needs. The Industrial Revolution, especially the process of mass production, created separate, well-barricaded groups.

The Customer, the Salesperson, the Planner, the Designer, the Worker, the Carrier and the Accountant were separated from each other. The Planner selected Customers. Salespersons determined Customer's needs. Designers created solutions to meet the Salesperson's determination of the Customer's needs. Workers built products according to the Designer's specifications. Carriers delivered the products to the Customer. Accountants handled payments and receipts.

Technological possibilities make this separatist philosophy obsolete. To meet world-wide competition, American organizations are incorporating "world class manufacturing" techniques [Schonberger, 1986] which promote collaboration and innovation to increase quality, flexibility and responsiveness.

Computer technologies allow small organizations to succeed in world competition. As companies re-establish close ties between producers and customers, they shift their focus away from "product features back to service to customers" [Hayes and Jaikumar, 1988, p.81].

In the 1980's, communication between customers and developers of complex systems remains separatist. Customer-developer interaction focuses on establishing *product features* rather than on *collaboration* and *service*. This focus increases costs, misunderstandings, and conflict during the system development process.

Disparate cultures, education, work situations, social status, and languages exacerbate communication difficulties between developers and users of computer systems. To make matters worse, programmers and system developers traditionally work alone, believing that independence is essential for creativity.

At the end of the 1980's, most systems developments fail. Social, technical and economic challenges of the 1990's are forcing us to find better ways of managing this increasingly difficult process.

The early test plan scenarios of this thesis encourage us to work together, while preserving our individual freedom to create--it is an effective tool for the 1990's and beyond.

BIBLIOGRAPHY

Abbott, R.J. and Moorhead, D.K. "Software Requirements and Specifications: A Survey of Needs and Languages." The Journal of Systems and Software. Vol 2., 1981, pp. 297-316.

Abelson, Robert P. "Psychological Status of the Script Concept." American Psychologist, July 1981, pp. 715-729.

Adelson, Beth. "When Novices Surpasses Experts: The Difficulty of a Task May Increase With Expertise." In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 55-67.

Adelson, Beth and Soloway, Elliot. "The Role of Domain Experience in Software Design." In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 233-251.

Agresti, William W. New Paradigms for Software Development. Washington: IEEE Computer Society Press, 1986.

Agusa, Kiyoshi, Kishimoto, Yoshinori and Ohno, Yutaka. "A Supporting System for Software Maintenance." In Teichroew, D. and G. David, eds. System Description Methodologies, eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 481-501.

Alavi, Maryam. "An Assessment of the Prototyping Approach to Information Systems Development." Communications of the ACM, Vol. 27, No. 6, pp. 556-563.

Alexander, H. "ECS - A technique for the formal specification and rapid prototyping of human-computer interaction." In People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986, pp. 157-179.

Alford, Mack. "SREM at the Age of Eight; The Distributed Computing Design System." Computer, April 1985, pp. 36-54.

Ambriola, Vincenzo and Notkin, David. "Reasoning About Interactive Systems." IEEE Transactions on Software Engineering, Vol. 14, No. 2, February 1988, pp. 272-276.

Anderson, Brian C. "A Comprehensive Statute of Limitations for Litigation Arising from Defective Custom Computer Systems." Stanford Law Review, July, 1985, pp. 1539-1572.

Anderson, Julie. "Tailored Applications." PC Tech Journal, Vol 6 No. 8, 1988, pp. 9-10.

Anderson, Nancy S. and Olson, Judith Reitman. eds. Methods For Designing Software to Fit Human Needs and Capabilities. Proceedings of the Workshop on Software Human Factors. Washington: National Academy Press, 1985.

Anderson, S. "Proving properties of interactive systems." In People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986, pp. 402-416.

Arango, Guillermo and Freeman, Peter. "Modeling Knowledge for Software Development." IEEE Software, Mar. 1985, pp. 63-66.

Attewell, Paul and Rule, James. "Computing and Organizations: What We Know and What We Don't Know." Communications of the ACM, Dec. 1984, Vol. 27, No. 12, pp. 1184-1192.

Avizienis, A. "Discussion Session 3: Reliability." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam:North-Holland, 1985, pp. 161-164.

Awani, Alfred O. Data Processing Project Management. Princeton: Petrocelli Books, 1986.

Babb, II, Robert G. "A Data Flow Approach to Unifying Software Specification, Design, and Implementation." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 9-13.

Balbin, I., Poole, P.C., and Stuart, C.J. "On The Specification and Manipulation of Forms." In Teichroew, D. and G. David, eds. System Description Methodologies, eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 239-252.

Balzer, R., Green, C. and Cheatham, T. "Software Technology in the 1990s Using a New Paradigm." Computer, Nov. 1983. pp. 39-45.

Balzer, R. et al., "Operational Specification as a Basis for Specification Validation," Theory and Practice of Software Technology. D. Ferrari, M. Bologani and J. Goguen, eds., North-Holland, Amsterdam, 1983.

Barlow, Kenneth J. "Effective Management of Engineering Design." Journal of Management in Engineering, Vol. 2, No. 1, Jan. 1985, pp. 51-66.

Basili, Victor R. "Changes and Errors as Measures of Software Development." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980a.

_______________. "Data Collection, Validation and Analysis." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980b.

_______________. "An Investigation of Human Factors in Software Development." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980c.

_______________. "Product Metrics." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980d.

_______________. "Quantitative Software Complexity Models: A Panel Summary." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980e.

_______________. "Resource Models." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York:IEEE Computer Society, 1980f.

_______________. "Tailoring SQA to fit your own life cycle." IEEE Software, March 1988, pp. 87-88.

_______________. "The TAME Project: Towards Improvement-Oriented Software Environments." IEEE Transactions on Software Engineering, Vol.14, No. 6, June 1988, pp. 758-773.

_______________. ed. Tutorial on Models and Metrics for Software Management and Engineering. Papers presented COMPSAC80. New York:IEEE Computer Society, 1980g.

Basili, Victor R. and Perricone, Barry T. "Software Errors and Complexity." Communications of the ACM, Jan 1984, Vol. 27 No. 1, Jan. 1984, pp. 42-52.

Basili, Victor R. and Rombach, H. Dieter. "Implementing Quantitative SQA: A Practical Model." IEEE Software, September 1987, pp. 6-8.

Basili, Victor R. and Weiss, D.M. "Evaluation of a Software Requirements Document by Analysis of Change Data." Proceedings of the 2nd International Conference on Software Engineering, March, 1981, pp. 314-324.

Basili, Victor R. and Zelkowitz, Marvin V. "Analyzing Medium-scale Software Development." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980.

Bate, R. R. and Ligler, G. T. "An Approach to Software Testing Methodology and Tools, " Proceedings of COMPSAC 78, November 13-16, 1978. Chicago, IL, pp. 476-480.

Beck, Larry. "Carousels Boost Productivity by 60%." Modern Materials Handling, February 1987, pp. 58-61.

__________. "Here's How We Stayed Within 10% of Budget." Modern Materials Handling, May 1986, pp. 63-64.

Berry, Daniel and Berry, Orna. "The Programmer-Client Interaction In Arriving at Program Specifications: Guidelines and Linguistic Requirements." In Teichroew, D. and G. David, eds. System Description Methodologies, eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 275-293.

Bersoff, Edward H. "Elements of Software Configuration Management." IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, Jan. 1984, pp. 79-87.

Bielski, J. P. and Blankertz, W. H. "The General Acceptance Test System (GATS)." Proceedings of COMPCON 77, Feb 28 - Mar 3, 1977, San Francisco, pp. 207-210.

Bigelow, Robert P. "The Challenge of Computer Law." Western New England Law Review. Vol. 7, Issue 3, 1985, pp. 397-404.

Birrell, N. D. and Ould, M. A. A Practical Handbook for Software Development. Cambridge: Cambridge University Press, 1985.

Bjorn-Andersen, Niels. In Human-Computer Interaction-INTERACT '84. Ed. B. Shackel. Amsterdam:Elsevier Science Publishers B.V., (North-Holland), 1985, pp. 839-846.

Blackler, F. H. M. and Brown, C. A. Job Redesign and Management Control: Studies in British Leyland and Volvo. Westmead, England: Saxon House, 1978.

Blum, Bruce I. "On How We Get Invalid Systems." In Third International orkshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 20-21.

____________. "A Paradigm for Developing Information Systems." IEEE Transactions on Software Engineering, Vol. SE-13, No. 4, April 1987, pp. 432-439.

Bockrath, Joseph. Dunham and Yount's Contracts, Specifications and Law for Engineers. New York: McGraw-Hill, Inc., 1986.

Bodart, Francois, Hennebert, Anne-Marie, Leheureux, Jean-Marie, Masson, Olivier, Pigneur, Yves. "DSL/DSA: A System For Requirements Specification, Prototyping and Simulation." In Teichroew, D. and G. David, eds. System Description Methodologies, eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 203-218.

Boehm, Barry W. "Industrial Software Metrics Top 10 List". IEEE Software. Sept. 1987, p. 84.

______________. "Software Design and Structure.", Horowitz, ed. Practical Strategies for Developing Large Software Systems. Reading, MA: Addison-Wesley, 1975.

______________. "Software Engineering." IEEE Transactions on Computers, Vol. C-25, No. 12, Dec. 1976, pp. 1226-1241.

______________. "Software Engineering Economics." IEEE Transactions on Software Engineering, Vol SE-10, No.1, Jan. 1984a, pp. 4-21.

______________. "Software and Its Impact: A Quantitative Assessment". DATAMATION, May, 1973, pp. 48-59.

______________. "Verifying and Validating Software Requirements and Design Specifications." IEEE Software, Jan. 1984b, pp. 75-88.

Boehm, Barry W., Brown, J.R., Lipow, M. "Quantitative Evaluation of Software Quality." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980.

Boehm, Barry W., Gray, Terrence E. and Seewaldt, Thomas. "Prototyping Versus Specifying:A Multiproject Experiment." In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 298-311.

Boehm, Barry W., Standish, Thomas A. "Software Technology in the 1990's: Using an Evolutionary Paradigm." Computer, Nov. 1983, pp. 30-37.

Boland, Richard J. "Control, Causality and Information System Requirements." Accounting, Organizations and Society, Vol. 4, No. 4, 1979, pp. 259-272.

Borgida, Alexander, Greenspan, Sol and Mylopoulos, John. "Knowledge Representation as the Basis for Requirements Specifications." Computer, April 1985, pp. 82-91.

Botting, Richard J. "An Eclectic Approach to Software Engineering." IEEE Software, Mar. 1985, pp. 25-29.

Boyd, Donald L. and Pizzarello, Antonio. "Introduction to the WELLMADE Design Methodology." IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978, pp. 276-282.

Branstad, Martha and Powell, Patricia B. "Software Engineering Project Standards." IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, Jan. 1984, 73-78.

Bratman, Harvey and Court, Terry. "The Software Factory." Computer, May 1975, pp. 28-37.

Brooke, J.B., and Duncan, K.D. "Experimental Studies of Flowchart Use at Different Stages of Program Debugging." In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985 pp. 335-364.

Brooks, Jr., Frederick P. "No Silver Bullet - Essence and Accidents of Software Engineering." Computer, April, 1987, pp. 10-19.

Brooks, Ruven E. "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology." In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 682-688.

Bryan, William and Siegel, Stanley. "Making Software Visible, Operational, and Maintainable in a Small Project Environment." IEEE Transactions on Software Engineering, Vol. SE-10, No.1, Jan. 1984, 59-67.

Buckley, Fletcher J. "Software Quality Assurance." IEEE Transactions On Software Engineering, Vol. SE-10, No. 1, Jan. 1984, pp. 36-41.

Burrell, Gibson and Morgan, Gareth. Sociological Paradigms and Organisational Analysis. Portsmouth, New Hampshire: Heinemann Educational Books Inc., 1985.

Card, David N., McGarry, Frank E. and Page, Gerald T. "Evaluating Software Engineering Technologies." IEEE Transactions on Software Engineering, Vol. SE-13, No. 7, July 1987, pp. 845-851.

Card, D. N. and Agresti, W. W. "Measuring Software Design Complexity." The Journal of Systems and Software 8, 1988, pp. 185-197.

Carpenter, M.B. and Hallman, H. K. "Quality Emphasis at IBM's Software Engineering Institute." IBM Systems Journal, Vol. 24, Nos. 3/4, 1985, pp. 121-133.

Carrier, Roger E., Hensey, Melville and Popovich, Rich. "Anatomy of a Successful Productivity Effort." Journal of Management in Engineering, Vol. 2, No. 1, Jan. 1985, pp. 47-55.

Carroll, J. M. and Thomas, John C. "Metaphor and the Cognitive Representation of Computing Systems." IEEE Transactions on Systems, Man and Cybernetics, Vol. 12, 1982, pp. 107-116.

Carroll, John M., Thomas, John C., Miller, Lance A., Friedman, Herman P. "Aspects of Solution Structure in Design Problem Solving." In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 272-283.

Carroll, John M., Thomas, John C. and Malhotra, Ashok. "Presentation and Representation in Design Problem Solving." In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 261-271.

Carroll, Paul B. "Computer Glitch - Patching Up Software Occupies Programmers and Disables Systems." Wall Street Journal, January 22, 1989, pp. 1, 13.

Cavano, Joseph P. and Frank S. LaMonica. "Quality Assurance in Future Development Environments." IEEE Software, September 1987, pp. 26-34.

Chandrasekharan, M., Dasarathy, B. and Kishimoto, Z.. "Requirements-Based Testing of Real-Time Systems: Modeling for Testability." Computer, April, 1985, pp. 71-80.

Chikofsky, Elliot J. "Application of an Information Systems Analysis and Development Tool to Software Maintenance." In Teichroew, D. and G. David, eds. System Description Methodologies, eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 503-516.

__________________. "Software Technology People Can Really Use." IEEE Software, March 1988, pp. 8-10.

Chikofsky, Elliot J. and Rubenstein, Burt L. "CASE:Reliability Engineering for Information Systems." IEEE Software, March 1988, p. 11-17.

Cho, Chin-Kuei. Quality Programming, Developing and Testing Software With Statistical Quality Control. New York: John Wiley & Sons, Inc., 1987.

Chretien-Dar, Barbara. "Uniform Commercial Code: Disclaiming the Express Warranty in Computer Contracts - Taking the Byte Out of the UCC." Oklahoma Law Review, Fall 1987, Vol. 40 No. 3, pp. 471-500.

Chu, Yaohan. Software Blueprint and Examples. Lexington, MA: Lexington Books, 1982.

Chumura, Louis J. and Ledgard, Henry F. Cobol With Style, Programming Proverbs. Rochelle, N.J.: Hayden Book Company, Inc, 1976.

Cohen, B., Harwood, W. T. and Jackson, M. I. The Specification of Complex Systems. Reading, MA: Addison-Wesley Publishing Company, 1986.

Collingridge, David. The Social Control of Technology. New York: St. Martin's Press, Inc., 1980.

Cox, Brad J. Object Oriented Programming, An Evolutionary Approach. Reading, MA: Addison Wesley Publishing Company, 1987.

Cunningham, R.J., Finkelstein, A., Goldsack, S., Maibaum, T. and Potts, C. In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 186-191.

Curtis, Bill. "By the Way, Did Anyone Study Any Real Programmers?" In Empirical Studies of Programmers. Eds. Soloway, Elliot and Sitharama Iyengar. Norwood, New Jersey: Ablex Publishing Corporation, 1986, pp. 256-262.

____________. "Foundations for a Measurement Discipline." IEEE Software, November 1987, pp. 89, 92.

Curtis, Bill, ed. Human Factors in Software Development. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985.

Dagwell, Ron, and Webber, Ron. "System Designers' User Models: A Comparative Study and Methodological Critique." Communications of the ACM, Nov. 1983, Vol. 26, No. 11, pp. 987-997.

Daly, Edmund B. "Management of Software Development." IEEE Transactions on Software Engineering. May 1977, pp. 229-242.

David, G. "Attributes of Sustem Design Methods." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam:North-Holland, 1985, pp. 333-338.

Davis, Alan M. "The Design of a Family of Application-Oriented Requirements Languages." Computer, May 1982, pp. 21-28.

deKleer, J. and Brown, J. S. "Assumptions and Ambiguities in Mechanistic Mental Models." In D. Gentner and A. S. Stevens, eds., Mental Models. Hillsdale, NJ: Lawrence Erlbaum, 1983.

Demetrovics, J., Knuth, E. and Rado, P. "Specification Metasystems." Computer, April 1982, pp. 20-35.

DeMillo, Richard A., McCracken, W. Michael, Martin, R. J., Passafiume, John F. Software Testing and Evaluation. Menlo Park: TheBenjamin/Cummings Publishing Company, Inc., 1987.

DeRemer, F. and Kron, H. H. "Programming-in-the-Large Versus Programming-in-the-Small." IEEE Trans. Software Engineering, June, 1976.

DeSena, Art. "Design for Testability - DOD Drafts Integrated Diagnostics Into Service." Computer Design, April 1, 1988, pp. 90-91.

Dietz, Jan. "Towards an Information System Development Environment." In Teichroew, D. and G. David, eds. System Description Methodologies, eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 27-34.

Digital Consulting Inc. Catalog of Seminars and Conferences. Andover, MA: Digital Consulting Inc., 1988.

Doherty, W. J. and Pope, W. G. "Computing as a Tool For Human Augmentation." IBM Systems Journal, Vol. 24, Nos. 3/4, 1985, pp. 306-320.

Dreyfus, Hubert L. and Dreyfus, Stuart E. Mind Over Machine, The Power of Human Intuition and Expertise in the Era of the Computer. New York: The Free Press, 1986.

Drucker, Peter. "Management and the World's Work." Harvard Business Review, September-October 1988, No.5, pp. 65-76.

Druffel, Larry E., Redwine, Jr., Samuel T. and Riddle, William E. "The STARS Program: Overview and Rationale." Computer, Nov. 1983, pp. 21-29.

Duffy, T.M., Curran, T. E. and D. Sass, D. "Document Design for Technical Job Tasks: An Evaluation." Human Factors, V. 25, 1983, pp. 143-160.

Dunham, J.R. and Druesi, E. "The Measurement Task Area." Computer, Nov. 1983, pp. 47-54.

Eastman, D.J. "The Game of the Gods: A Mosaic of Views on Systems Development." Research Paper for PhD Studies. 1987.

Eastman, D.J. "The Struggle to Automate: A Case Study." Research Paper for PhD Studies. 1986.

Eberhart, Jonathan. "The Sayings of Science." Science News, Vol.133, January 30, 1988, pp. 72-73.

Elden, Max. "Sociotechnical Systems Ideas as Public Policy in Norway: Empowering Participation Through Worker-Managed Change." The Journal of Applied Behavioral Science, Vol. 22, No. 3, 1986, pp. 239-255.

Endres, Albert. "An Analysis of Errors and Their Causes in System Programs." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V. R. Basili. New York: IEEE Computer Society, 1980.

Epple, Wolfgang K. and Koch, Guenter R. "Specification of Process Control Systems with SATS." In Teichroew, D. and G. David, eds. System Description Methodologies, eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 141-155.

Essink, Leo J.B. "A Modelling Approach to Information System Development." In Information Systems Design Methodologies: Improving the Practice. Eds. T. W. Olle, H. G. Sol and A. A. Verrijn-Stuart. Amsterdam: Elsevier Science Publishers B. V., 1986, pp. 55-84.

Estrin, G. "The Story of SARA." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam: North-Holland, 1985, pp. 29-46.

Evans, Michael W. and Marciniak, John J. Software Quality Assurance and Management. New York: John Wiley & Sons, 1987.

Eveking, H. "A Structured Methodology for the Verification of Multilevel Hardware Descriptions." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam: North-Holland, 1985, pp. 71-86.

Felker, D. C., ed. Document Design: A Review of the Relevant Research. American Institute for Research. Technical Report 75002-4/80, Washington, D.C., 1980.

Fickas, Stephen and Nagarajan, P. "Critiquing Software Specifications." IEEE Software, November 1988, pp. 37-47.

Fitter, M. and Green, T. R. G. "When do Diagrams make Good Computer Languages?" In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 374-395.

Floyd, Christiane. "On the Relevance of Formal Methods to Software Development." In Formal Methods and Software Development, eds. Hartmut Ehrig, Christiane Floyd, Maurice Nivat and James Thatcher. Berlin: TAPsoft, 1985, pp. 1-11.

Forman, Fred L. and Hess, Milton S. "Form Precedes Function." Computerworld, September 5, 1988, pp. 65-70.

Frank, Geoffrey A., Redwine, Jr., Samuel T. and Squires, Stephen L. "The Systems Task Area." Computer, November, 1983, pp. 71-76.

Fraser, S.D. and Silvester, P. P. "An Interactive Empirical Approach to the Validation of Software Package Specifications." In Third International Workshop on Software Specification and Design. Washington, D. C.: IEEE Computer Society Press, 1985, pp. 60-62.

Freed, Roy N. "Products Liability in the Computer Age." Jurimetrics Journal, Summer, 1977, pp. 270-285.

Fried, Charles. Contract as Promise A Theory of Contractual Obligation. Cambridge: Harvard University Press, 1981.

Fryer Sandra R. "Reviewing Software Specifications." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 67-69.

Gehani, Narain and McGettrick, Andrew. Software Specification Techniques. Reading, MA: Addison-Wesley Publishing Company, 1986.

Giese, Clarence. "Partitioning Considerations for Complex Computer Based Weapon Systems." The Journal of Systems and Software. Vol. 1, 1979, pp. 3-18.

Gilb, Tom. "The Pre-Natal Death of the CIS Project:A Software Disaster Story." The Journal of Systems and Software 8, 1988, pp. 161-163.

Giloi, W.K. and Shriver, B. D. "An Overview of the Computer System Design Process." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam:North-Holland, 1985, pp. 1-10.

Glass, Robert L. "Software Productivity Improvement: Who's Doing What?" The Journal of Systems and Software 8, 1988, pp. 159-160.

Goddard, Walter E. "How to use `*People Power*' to Upgrade Your Operations." Modern Materials Handling, December 1987, pp. 67-69.

Gold, Bela. "Computerization in Domestic and International Manufacturing." California Management Review, Winter, 1989, Vol. 31, No. 2, p. 129-143.

Goldberg, R. "Software Engineering: An Emerging Discipline." IBM Systems Journal, Vol. 25, 1986, pp. 334-353.

Goos, G. and Hartmanis, J. eds. Lecture Notes in Computer Science 186 Formal Methods and Software Development. Heidelberger: Springer-Verlag, 1985.

Gordon, Mark L. and Starr, Steven B. "Emerging Issues in Computer Procurements." Computer Law Journal, Vol. VI, No. 1, Summer 1985, pp. 119-142.

____________________________________. "Software Development Contracts and Consulting Arrangements: A Structure for Enforceability and Practicality." Western New England Law Review, Vol. 7, Issue 3, 1985, pp. 487-853.

Gould, John D. and Lewis, C. "Designing for Usability of Key Principles and What Designers Think." Communications of the ACM 28. New York: Association of Computing Machinery, 1985, pp. 300-311.

Gould, John D. and Boies, Stephen J. "Human Factors Challenges in Creating a Principal Support Office System--The Speech Filing System Approach." ACM Transactions on Office Information Systems, Vol. 1, No. 4, October 1983, pp. 273-298.

________________. "Speecch Filing System." IBM Systems Journal, Vol. 23, No. 1, 1984, pp. 65-66.

Gould, L. and Finzer, W. "Programming by Rehearsal." Byte, June, 1984, pp. 187-210.

Grady, Robert B. "Measuring and Managing Software Maintenance." IEEE Software, September 1987, pp. 35-45

Gray, David. "The Formal Specification of a Small Bookshop Information System." IEEE Transactions on Software Engineering, Vol. 14, No. 2, February 1988, pp. 263-272.

Gruman, Galen. "Software engineers face fundamental challenges." IEEE Software, November 1987, pp. 93-96.

Gueth, R. and Kriz, J. "Semiformal Specification and Design of Multilevel Computer Systems." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam:North-Holland, 1985, pp. 273-290.

Haas, Volkmar H. and Koch, Gunter R. "Application-Oriented Specifications. Computer, May 1982, pp. 10-11.

Hamilton, M. and Zeldin, S. "The Relationship Between Design and Verification." The Journal of Systems and Software, Vol. 1, 1979, pp. 29-56.

Hanau, P. R. and Lenorovitz, D. R. "A Prototyping and Simulation Approach to Interactive Computer System Design." in Proceedings of the 17th Design Automation Conference, Minneapolis, Minn. 1980, pp. 23-25.

Haney, F. M. "What It Takes to Make MAC, MIS and ABM Fly". DATAMATION, June, 1974, pp. 168-169.

Harrison, M.D. and Monk, A. F. eds. People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986.

Hashimoto, Keiji. "System Analysis by Extended Data Flow Diagram with Events and Timing.' In Proceedings COMPSAC87. New York: Computer Society Press, 1987, pp. 117-123.

Hayes, Robert H. and Jaikumar, Ramchandran. "Manufacturing's Crisis: New Technologies, Obsolete Organizations." Harvard Business Review, September-October 1988, No.5, pp. 77-85.

Head, R. V. "Automated System Analysis." DATAMATION, August 15, 1971, pp. 22-24.

Hein, K. P. "Information System Model and Architecture Generator." IBM Systems Journal, Vol. 24, Nos. 3/4, 1985, pp. 213-235.

Heninger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications." IEEE Transactions of Software Engineering, Vol. SE-6(1), Jan 1982, pp. 2-12.

Henry, G. G. "IBM Small-System Architecture and Design--Past, Present and Future." IBM Systems Journal, Vol. 25, 1986, pp. 321-333.

Hewett, T. T. "The Role of Iterative Evaluation in Designing Systems for Usability." In People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986, pp. 196-214.

Hirsch, P. "GAO Hits Wimmix Hard; FY'72 Funding Prospects Fading Fast." DATAMATION, March 1, 1971, p. 41.

Hirschhein, R., Land, F. and Smithson, S. In Human-Computer Interaction-INTERACT '84. Ed. B. Shackel. Amsterdam: Elsevier Science Publishers B.V., (North-Holland), 1985, pp. 855-860.

Hoeker, D.G. and Pew, R. W. "User Input to the Design and Evaluation of Computer-Assisted Service Delivery." Report # 4358. Cambridge: Bolt Beranek and Newman, Inc., 1980.

Hoffnagle, G. F. and Beregi, W. E. "Automating the Software Development Process." IBM Systems Journal, Vol. 24, Nox. 3/4, 1985, pp. 102-120.

Howden, William E. "The Theory and Practice of Functional Testing." IEEE Software. September 1985, pp. 6-17.

Humphrey, Watts S. "Characterizing the Software Process: A Maturity Framework." IEEE Software, March 1988, pp. 73-79.

_________________. "The IBM Large-Systems Software Development Process: Objectives and Direction." IBM Systems Journal, Vol. 24, Nos. 3/4, 1985, pp. 76-78.

_________________. Managing for Innovation: Leading Technical People, Englewood Cliffs, N.J.: Prentice-Hall, 1987.

Humphrey, Watts S. and Kitson, D. H. "A Method for Assessing the Software Engineering Capability of Contractors." Tech. Report SEI-87-TR-23, Pittsburgh: Software Engineering Institute, Sept. 1987.

Hymowitz, Carol. "One Firm's Bid to Keep Blacks, Women." Wall Street Journal, February 16, 1989, p. B1.

Ikadai, Mieko, Ikadai, Yukio and Nobe, Yoshikazu. "Requirements Specification with M-Box." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 109-113.

"Increasing Productivity With Integrated Handling Systems." Modern Materials Handling, Dec. 10, 1984, pp. 53-57.

Information Systems Design Methodologies: A Comparative Review. Olle, E. W., Sol, H. G. and Verrijn-Stuart, A. A., eds. Amsterdam: North-Holland, 1982.

Jacob, Robert J.K. "Using Formal Specifications in the Design of a Human-Computer Interface." Communications of the ACM, April 1983, Vol. 26, No. 4, pp. 259-264.

Jick, T. D. "Mixing Qualitative and Quantitative Methods: Triangulation in Action." In Method and Analysis in Organizational Research, eds. Bateman, Thomas S. and Gerald R. Ferris. Reston, Virginia: Reston Publishing Company, Inc., 1984, pp. 35-42.

Jones, J. "MacCadd - an enabling software method support tool." In People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986, pp. 132-154.

Kalinsky, David and Levy, Eyal. "A Functional Approach to Software Specification and Design." In Third International Workshop on Software Specification and Design. Washington, D. C.: IEEE Computer Society Press, 1985, pp. 118-119.

Kant, Elaine and Newell, Allen. "Problem Solving Techniques for the Design of Algorithms." In Human Factors in Software Development, ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 211-232.

Kenfield, Dexter L. "Remedies in Software Copyright Cases." Computer Law Journal, Vol VI, 1985, pp. 1-33.

Kishida, Kouichi, Teramota, Masanori, Torii, Koji and Urano, Yoshiyori. "Quality-Assurance Technology in Japan." IEEE Software, September 1987, pp. 11-17.

Kiss, G. and Pinder, R. "The Use of Complexity Theory in Evaluating Interfaces." In People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986, pp. 447-463.

Klein, Henry. "Methodology for System Description Using the Software Design & Documentation Language." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 111-138.

Kloster, Gregory and Tischer, Kristine. "Man-Machine Interface Design Process." In Human-Computer Interaction-INTERACT '84. Ed. B. Shackel. Amsterdam: Elsevier Science Publishers B.V., North-Holland, 1985, pp. 889-894.

Koomen, C. J. "From Specification Towards Implementation - A Methodological Viewpoint." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam: North-Holland, 1985, pp. 105-122.

Koretz, Gene. "How the Hispanic Population Boom Will Hit the Workforce." Business Week, February 20, 1989, p. 21.

Knirk, Dwayne. "Specialty." The Letter T, Vol. 2, No. 1, January 1988, pp. 3-4.

Krasner, Herb. "Modeling Software Systems as Unified Views." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 122-124.

Krepchin, Ira P. "The Human Side of Competitiveness." Modern Materials Handling, February 1988, pp. 64-67.

_______________. "We Simulate All Major Projects." Modern Materials Handling, August 1988, pp. 83-88.

Kruesi, Elizabeth. "The Human Engineering Task Area." Computer. November 1983, pp. 86-93.

Lampson, Butler W. "Hints for Computer System Design." IEEE Software. January 1984, pp. 11-28.

Lauber, Rudolf. "Development Support Systems." Computer, May 1982, pp. 36-46.

LeFevre, Karen Burke. Invention as a Social Act. Carbondale and Edwardsville: Southern Illinois University Press, 1987.

Lehman, J.D. and Yavneh, N. "The Total Life Cycle Model." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 135-138.

Lehman, M. M. "Program Evaluation." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 3-25.

Leonard, Jonathan S. "The Changing Face of Employees and Employment Regulation." California Management Review, Winter, 1989, Vol. 31, No. 2, p. 29-38.

Levene, A. A. and Mullery, G. P. "An Investigation of Requirement Specification Languages: Theory and Practice." Computer. Vol 15(5), May, 1982, pp. 50-59.

Leventhal, Naomi S. "Systems Designing-What is Wrong?" MIS Week, March 10, 1986, pp. 48-49.

Levidow, Les and Young, Bob, ed. Science, Technology and the Labour Process. Atlantic Highlands, NJ: CSE Books, 1981.

Levin, Sandra J. "Examining Restraints on Freedom to Contract as an Approach to Purchaser Dissatisfaction in the Computer Industry." California Law Review. Vo. 74, No. 6, December, 1986.

Linn, J. L. "Discussion Session 2: Language Issues in System Design Methods." In Methodologies For Computer System Design. Eds. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam: North-Holland, 1985, pp. 123-126.

Linnell, Dennis. "SAA: IBM's Road Map to the Future." PC Tech Journal, Vol. 6 No. 4, April 1988, pp. 86-105.

Lipow, Myron. "Prediction of Software Failures." The Journal of Systems and Software. Vol. 1, 1979, pp. 71-75.

Lissandre, Michel, Lagier, Pierre and Skalli, Ahmed. "SAS - A Specification Support System." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 221-238.

Long, J. B. People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986.

Lucas, Henry C. A Casebook For Management Information Systems. New York: McGraw-Hill, 1986.

______________. Why Information Systems Fail. New York: Columbia University Press, 1975.

Ludewig, Jochen Ludewig. "Computer-Aided Specification of Process Control Systems." Computer, May 1982, pp. 12-20.

______________________. "A Note on Abstraction in Software Descriptions." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 535-543.

Lundberg, Bengt. "On Relative Strength of Information Models." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 403-415.

Lundberg, Craig C. "Hypothesis Creation in Organizational Behavior Research." In Method and Analysis in Organizational Research. Eds. Bateman, Thomas S. and Gerald R. Ferris. Reston, Virginia: Reston Publishing Company, Inc., 1984, pp. 35-42.

Luqi and Ketabchi, Mohammad. "A Computer-Aided Prototyping System IEEE Software, March 1988, pp. 66-71.

Lynch, Bruce W. "Mapping the Program Development Cycle." Programmer's Update, Vol. 6, No. 3, July/August 1988, pp. 62-63.

MacLean, A., Barnard, P., and Wilson, M. "Rapid Prototyping of Dialogue for Human Factors Research: the EASIE approach." In People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986, pp. 180-195.

Macro, Allen and John Buxton. The Craft of Software Engineering. Reading, MA: Addison-Wesley Publishing Company, 1987.

Maiocchi, Marco. "The Use of Petri Nets in Requirements and Functional Specification." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 253-274.

Mann, Nancy R. "Why It Happened in Japan and Not in the U.S." Chance: New Directions for Statistics and Computing, Summer, 1988, Vo. 1, No. 3, p. 8-15.

Mantha, Robert W. "Data Flow and Data Structure Modeling for Database Requirements Determination: A Comparative Study." MIS Quarterly, Dec. 1987, pp. 531-545.

Marca, David A. and Cashman, Paul M. "Towards Specifying Procedural Aspects of Cooperative Work." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 151-154.

Marca, David A. and McGowan, Clement L. SADT, Structured Analysis and Design Technique. New York: McGraw-Hill Book Company, 1988.

Marcellino, James J. "Expert Witnesses in Software Copyright Infringement Actions." Computer Law Journal, Vol. VI, 1985, pp. 35-54.

Marcellino, James J. and Kenfield, Dexter L. "How to Handle Copyright Suits over Infringement of Software." Boston Bar Journal, Vol. 29, No. 5, Nov./Dec., 1985, pp. 12-14.

Markus, M. Lynne and Bjorn-Andersen, Niels. "Power Over Users: Its Exercise By System Professionals." Communications of the ACM, Vol. 30, No. 6, June 1987, pp. 498-504.

Marmor-Squires, Ann B., Riddle, William E., Sumrall, George E. and Wileden, Jack C. "The Support Systems Task Area." Computer, Nov. 1983, pp. 97-104.

Mason, Paul E. and Foley, Denise. "International Protection of Computer Software: The Japanese Experiment." Boston Bar Journal, Vol. 29, No. 5, Nov./Dec., 1985, pp. 28-34.

Mason, R. E. A. "Concrete Use of Abstract Development Formalisms." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 75-89.

Martin, James and McClure, Carma. Structured Techniques: The Basis for CASE. Englewood Cliffs, NJ: Prentice Hall, 1988.

Mathiassen, Lars and Munk-Madsen, Andreas. "Formalization in Systems Development" In Formal Methods and Software Development, eds. Hartmut Ehrig, Christiane Floyd, Maurice Nivat and James Thatcher. Berlin: TAPsoft, 1985, pp. 101-116.

Matsumura, Kazuo, Mizutani, Hiroyuki and Arai, Masahiko. "An Application of Structural Modeling to Software Requirements Analysis and Design." IEEE Transactions on Software Engineering, Vol. SE-13, No. 4, April 1987, pp. 461-471.

Mayer, Richard E., "Comprehension as Affected by Structure of Problem Representation." In Human Factors in Software Development. Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 320-326.

Mays, R. G., Orzech, L. S., Ciarfella, W. A. and Phillips, R. W. "PDM: A Requirements Methodology for Software System Enhancements." IBM Systems Journal, Vol. 24, Nos. 3/4, 1985, pp. 134-149.

McAllister, A. J. "Metasystem Support for Specification Environments." Tech. Report 88-1, Computational Science Dept., Univ. of Saskatchewan, Saskatoon, Sask., Canada, Jan. 1988.

McCoy, Charles F. and Schmitt, Richard B. "United Education's Computer Blunders Form Vortex of Big Student Loan Fiasco." Wall Street Journal, Marcch 10, 1989, p. A6.

McClure, Carma. "The CASE for Structured Development." PC Tech Journal, Vol. 6, No. 8, pp. 51-67.

McGill, James P. "The Software Engineering Shortage: A Third Choice." IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, Jan. 1984, pp. 42-49.

Melichar, Paul R., Management Strategies for High-Risk Project. Chicago: IBM Systems Science Institute, 1980.

Methodologies For Computer System Design, Proceedings of the IFIP WG 10.1 Workin Conference on Methodologies for Computer System Design,Lille, France, 15-17 September, 1983. Amsterdam: North-Holland, 1985.

Meyer, Bertrand. "On Formalism in Specifications." IEEE Software. January 1985, pp. 6-26.

Miller, D.C. and Pew, R. W. "Exploiting User Involvement in Interactive System Development." In Proceedings of the Human Factors Society Annual Meeting. Santa Monica, CA: Human Factors Society, 1983, pp. 45-49.

Miller, L. A. "Natural Language Programming: Styles, Strategies and Contrasts." In Human Factors in Software Development. Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 400-431.

Mills, Harlan D. "Software Development." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V. R. Basili. New York: IEEE Computer Society, 1980.

Mills, Harlan D., Dyer, Michael and Linger, Richard C. "Cleanroom Software Engineering." IEEE Software, September 1987, pp. 19-25.

Mills, Miriam K., "Luddites at the Terminal and Other Renegades." Rutgers Computer & Technology Law Journal, Vol. 10, No. 1, 1983, pp. 115-126.

Mislow, Christopher M. "Reducing the HIgh Risk of High Tech: Legal Planning for the Marketing of Computer Systems." American Business Law Journal, Vol. 23/1, Spring 1985, pp. 123-139.

Mizuno, Y. "Software Quality Improvement." Computer, March 1983, pp. 66-72.

Moor, James H. "What is Computer Ethics?" Metaphilosophy. Vol. 16, No. 4, October 1985, pp. 266-275.

Morgan, Gareth. Images of Organization. Beverly Hills: Sage Publications, 1986.

Morris, Daniel A. "Contracting for the Acquisition of Computer Hardware and Software." Case & Comment, Nov./Dec. 1985, Vol. 91, No. 6, pp. 32-41.

Murphy, Kevin J. "Listening Makes a Difference." Modern Materials Handling, March, 1988, p. 43.

Narayanaswamy, K. and Scacchi, Walt. "A Database Foundation to Support Software System Evolution." The Journal of Systems and Software, Vol. 7, 1987, pp. 37-49.

Naumann, J. D., Davis, G. B. and McKeen, J. D. "Determining Information Requirements: A Contingency Method for Selection of A Requirements Assurance Strategy." Journal of Systems and Software, Vol.1(4), 1980, pp. 273-282.

Necco, Charles R., Gordon, Carl L. and Tsai, Nancy. "Systems Analysis and Design: Current Practices." MIS Quarterly, Dec. 1987, 461-476.

Neches, Robert, Balzer, Bob, Goldman, Neil, Wile, David. "Transferring Users' Responsibilities to a System: The Information Management Computing Environment." In Human-Computer Interaction-INTERACT '84. Ed. B. Shackel. Amsterdam: Elsevier Science Publishers B.V., North-Holland, 1985, pp.421-426.

Nelson, R. Ryan. "Training End Users: An Exploratory Study." MIS Quarterly, Dec. 1987, pp. 547-559.

Norcio, A. F. and Chmura, L. J. "Designing Complex Software." The Journal of Systems and Software 8, 1988, pp. 165-184.

Nord, Melvin. Legal Problems in Engineering. New York: Chapman & Hall, Limited, 1956.

Norman, Colin. "DOD Lists Critical Technologies." Science, March March 24, 1989, Vol. 243, p. 1543.

Norman, Donald A. and Draper, Stephen W. User Centered System Design, New Perspectives on Human-Computer Interaction. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers, 1986.

Oglesby, Charles E. and Urban, Joseph E. "The Human Resources Task Area." Computer, Nov. 1983, pp. 65-70.

Olive, Antoni. "Conceptual Languages for Information Systems Modelling." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 389-401.

Olle, T. W., Sol, H. G. and Verrijn-Stuart, A. A. Information Systems Design Methodologies: Improving the Practice. Amsterdam: North-Holland, 1986.

Osgood, C. E., Suci, G. J. and Tannenbaum, P. J. The Measurement of Meaning. Champaign-Urbana: University of Illinois Press, 1957.

Osterweil, Leon J., Brown, John R. and Stucki, Leon G. "ASSET: A Life Cycle Verification and Visibility System." The Journal of Systems and Software. Vol. 1, 1979, pp. 77-86.

Ostrand, Thomas J. "The Use of Formal Specifications in Program Testing." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 253-255.

Pacini, G. and Turini, F. "Animation of Software Requirements." Industrial Software Technology. Ed. R.J. Mitchel. London: P. Peregrinus Ltd., 1987.

Parnas, D. L. and Weiss D. M. "Active Design Reviews: Principles and Practices." Journal of Systems and Software, 7, 1987, pp. 259-265.

Parr, F. N. "An Alternative to the Rayleigh Curve Model for Software Development Effort." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980.

Perardi, F., Cerchio, L., Scrignaro, D. and Modesti, M. "ISDN: A System Designed According to an SDL Based Methodology." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 561-576.

Perrolle, Judith A. Computers and Social Change. Belmont, CA: Wadsworth Publishing Company, 1987.

Petersen, I. "The Complexity of Computer Security." Science News, Vol. 134. No. 13, September 24, 1988, p. 199.

Petschenik, Nathan H. "Practical Priorities in System Testing." IEEE Software, September 1985, pp. 16-23.

Phan, Dien, Vogel, Douglas and Nunamaker, Jay. "The Search for Perfect Project Management." Computerworld, September 26, 1988, Vol. XXII, No.39, pp. 95-100.

Poston, Robert. "Looking for a Test Comprehensiveness Measure." IEEE Software, November 1985, p. 78.

______________. "Preventing the Most-Probable Errors in Testing." IEEE Software, July 1987, pp. 86-88.

______________. "Preventing the Most-Probable Errors in Testing." IEEE Software, September 1987, pp. 81-83.

______________. "Preventing the Most-Probable Errors in Testing." IEEE Software, March 1988, pp. 86-88.

______________. "Software Test Coverage Measure Automated in T." The Letter T. Vol. 2, No. 2, April 1988, pp. 2-3.

______________. "Using T to Get Requirements Right." The Letter T. Vol. 2, No. 3, August 1988, pp. 2-3.

______________. "Workaday Software Requirements Specifications." IEEE Software, September 1985, pp. 63-65.

Poston, Robert M. and Knirk, Dwayne L. "PEI Testing Methodology, Rev.3," Tech. report, Programming Environments, Inc., Tinton Falls, N.J., June 1986.

Poston, Robert M. and Bruen, Mark W. "Counting Down to Zero Software Failures." IEEE Software, September, 1987, pp. 54-61.

Potosnak, Kathleen. "Creating Software that People Can and Will Use." IEEE Software, Sept. 1987, pp. 86-87.

__________________. "Recipe for a Usability Test." IEEE Software, November 1988, pp. 89-90.

__________________. "Setting Objectives for Measurably Better Software." IEEE Software, March 1988, pp. 89-90.

__________________. "Where Human Factors Fits in the Design Process." IEEE Software, November 1987, pp. 90-91.

Puncello, P. Paolo, Torrigiani, Piero, Pietri, Francesco, Burlon, Riccardo, Cardile, Bruno and Conti, Mirella. "ASPIS: A Knowledge-Based CASE Environment." IEEE Software, March 1988, p. 58-65.

Putnam, Lawrence H. "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V.R. Basili. New York: IEEE Computer Society, 1980.

Quirk, W. J., ed. Verification and Validation of Real Time Software. Berlin: Springer-Verlag, 1985.

Radice, R.A., Harding, J. T., Munnis, P. E., Phillips, R. W. "A Programming Process Study." IBM Systems Journal, Vol. 24, Nos. 3/4, 1985, pp. 91-101.

Radice, R. A., Roth, N. K., O'Hara, Jr., A. C., Ciarfella, W. A. "A Programming Process Architecture." IBM Systems Journal, Vol. 24, Nos. 3/4, 1985, pp. 79-90.

Ramanathan, Jayashree and Sarkar, Soumitra. "Providing Customized Assistance for Software Lifecycle Approaches." IEEE Transactions On Software Engineering, Vol. 14, No. 6, June 1988, pp. 749-757.

Ramsey, H. Rudy, Atwood, Michael E. and Van Doren, James R. "Flowcharts Versus Program Design Languages: An Experimental Comparison." In Human Factors in Software Development. Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 315-319.

Ramsey, H. Rudy., Atwood, M. E. and Willoughby, J. K. "Paper Simulation Techniques in User Requirements Analysis for Interactive Computer Systems." In Proceedings of the Human Factors Society Annual Meeting. Santa Monica, CA: Human Factors Society, 1979, pp. 64-68.

Ratcliff, Bryan and Siddiqi, Jawed I. A. "An Empirical Investigation into Problem Decomposition Strategies used in Program Design." In Human Factors in Software Development. Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 284-297.

Reich, Robert B. "Teaching to Win in the New Economy." Technology Review, August/September, 1988, pp. 24-25.

Reifer, Donald J. "Specification and Design Technology: Is Enough Enough?" In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, p. 192.

Richter, Charles. "Toward More Exploration in Large-Scale Software Development." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 193-194.

Rifkin, Glenn. "MIS Education Assumes a Foreign Accent." Computerworld, March 13, 1989. Vol. XXIII, No. 11, pp. 1, 66.

Rigo, J. T. "How to Prepare Functional Specifications." DATAMATION, May 1974, pp. 78-80.

Rodau, Andrew. "Computer Software: Does Article 2 of the Uniform Commercial Code Apply?" Emory Law Journal, Vol. 35, No. 3, Summer, 1986, pp. 853-920.

Rodriguez, Jorge E. and Greenspan, Sol J. "Directed Flowgraphs: The Basis of a Specification and Construction Methodology for Real-Time Systems". The Journal of Systems and Software. Vol. 1, 1979, pp. 19-27.

Roman, Gruia-Catalin. "A Taxonomy of Current Issues in Requirements Engineering." Computer, April 1985, pp. 14-22.

Rosenblum, Howard S. "Software Escrows: Legal Concerns." Boston Bar Journal, Vol. 29, No. 5, Nov./Dec., 1985, pp. 22-27.

Ross, Douglas T. "Applications and Extensions of SADT." Computer, April 1985, pp. 25-33.

_______________. An Introduction to SADT^{tm} Structured Analysis and Design Technique. Waltham, MA: Softech, Inc., 1976.

_______________. "Structured Analysis: A Language for Communicating Ideas." IEEE Software, Jan. 1977, pp. 16-34.

Ross, Douglas T., Goodenough, J. B. and Irvine, C. A. "Software Engineering: Process, Principles and Goals." Computer, May 1975, pp. 17-27.

Ross, Douglas T. and Shoman, K. E. "Structured Analysis for Requirements Definition." IEEE Trans. Software Engineering, Jan. 1977, pp. 6-15.

Ruskin, Arnold M. and Estes, W. Eugene. "Organizational Factors in Project Management." Journal of Management in Engineering, Vol. 2, No. 1, Jan. 1985, pp. 3-9.

Rzepka, William and Ohno, Yutaka. "Requirements Engineering Environments: Software Tools for Modeling User Needs." Computer, April 1985, pp. 9-12.

Saracco, R., Cerchio, L. and Bagnoli, P. L. "Specification and Design Methodologies." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 35-44.

Schacter, Esther Roditti. "Techniques for Litigation Avoidance in Contracting for Computer Systems." Rutgers Computer & Technology Law Journal, Vol. 10, No. 1, 1983, pp. 109-114.

Scharer, Laura. "The Prototyping Alternative." ITT Programming, Vol. 1, No. 1, 1983, pp. 34-43.

Schleifer, Liane A. "Damage Awards and Computer Systems--Trends." Emory Law Journal, Vol. 35, No. 1, Winter, 1986, pp. 255-290.

Schlender, Brenton R. "Missed Deadline: Its Failure to Deliver On Promised Software Hits Microsoft Hard." March 8, 1989, p. A1, A6.

Schmidt, E. Controlling Large Software Development in a Distributed Environment, Ph.D. thesis, University of California, Berkeley, 1982.

Schneiderman, Ben. "Control Flow and Data Structure Documentation: Two Experiments." In Human Factors in Software Development. Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 365-372.

Schonberger, Richard J. World Class Manufacturing - The Lessons of Simplicity Applied. New York: The Free Press, 1986.

Schoumacher, Bruce. Engineers and the Law, An Overview. New York: Van Nostrand Reinhold Company, 1986.

Schrage, Michael. "Customers May Be Your Best Collaborators." Wall Street Journal, February 27, 1989, p. A10.

Schrama, C.J. "Computer Aided Requirements Gathering and Prototyping." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 103-109.

Schwartz, Felice N. "Management Women and the New Facts of Life." Harvard Business Review, January-February, 1989, Number 1, p. 67-76.

Second Software Engineering Standards Application Workshop. Silver Spring, MD: IEEE Computer Society Press, 1983.

Seewaldt, T. and Estrin, G. "A Multilevel Design Procedure to Foster Functional and Informational Strength." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam: North-Holland, 1985, pp. 11-28.

Selby, Richard W. "Cleanroom Software Development: An Empirical Evaluation." IEEE Transactions on Software Engineering. Vol. SE-13, No. 9, September 1987, pp. 1027-1037.

Shaw, Mary. "What Can We Specify? Issues in the Domains of Software Specifications." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 214-215.

Shen, Vincent. "A Little Quality Time." IEEE Software, September 1987, p. 84.

Shemer, Itzhak. "Systems Analysis: A Systemic Analysis of a Conceptual Model." Communications of the ACM, Vol. 30, No. 6, June 1987, pp. 506-512.

Sheppard, Sylvia B., Kruesi, Elizabeth and Curtis, Bill. "The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications." In Human Factors in Software Development. Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 327-334.

Sievert, Gene E. and Mizell, Terrence A. "Specification-Based Software Engineering with TAGS." Computer, Aprio 1985, pp. 56-65.

Skwirzynski, Jozef K. Software System Design Methods, The Challenge of Advanced Computing Technology. Heidelberg: Springer-Verlag, 1985.

Slaughter, Gary. The Team Approach to DP Productivity (TAP). Bethesda: Gary Slaughter Corporation, 1980.

Sloman, A. "What are the Aims of Science?" Radical Philosophy, No. 13, 1976, pp. 7-17.

Smedinghoff, Thomas J. The Legal Guide to Developing, Protecting, and Marketing Software, Dealing with Problems Raised by Customers, Competitors and Employees. New York: John Wiley & Sons, 1985.

Smolenski, R. J. "Test Plan Development." Journal of System Management, Vol. 32(2), February 1981, pp. 32-37.

Software: An Emerging Industry. Paris: Organisation for Economic Co-operation and Development, 1985.

Soloway, Elliot and Iyengar, Sitharama. Empirical Studies of Programmers. Norwood, N.J.: Ablex Publishing Corporation, 1986.

Soloway, Elliot and Ehrlich, Kate. "Empirical Studies of Programming Knowledge." In Human Factors in Software Development. Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 68-82.

Sorenson, Paul G., Tremblay, Jean-Paul and McAllister, Andrew J. "The Metaview System for Many Specification Environments." IEEE Software, March 1988, pp. 30-38.

Sorenson, Paul G. and Tremblay, Jean-Paul. "SPSL/SPSA:A Minicomputer Database System for Structured Systems Analysis and Design." Proc. ACM SIG-Small Database Systems. ACM: New York, 1981, pp. 109-118.

Swanson, E. Burton and Beath, Cynthia M. "The Use of Case Study Data in Software Management Research." The Journal of Systems and Software, Vol. 8, 1988, pp. 63-71.

Stankovic, John. "A Technique to Identify Implicit Information Associated with Modified Code." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 457-480.

Stapleton, Lisa and Puttre, Michael. "NASA's MIS Crashes" Information Week, March 21, 1988, pp. 8-10.

Teichroew, D. "PSL/PSA -- A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." Proceedings of Second International Conference on Software Engineering, 1976.

Teichroew, D. and David, G., eds. System Description Methodologies. Amsterdam: North-Holland, 1985.

Teichroew, D. and Sayani, H. "Automation of System Building." DATAMATION, August 15, 1971, pp. 25-30.

Teitelman, W. "Do What I Mean: The Programmer's Assistant." Computers and Automation, 1972, V. 21(4) pp. 8-11.

Thomas, R. J. "Systems Education." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 615-616.

Topper, Andrew. "Excelling with CASE." PC Tech Journal, August 1988, pp. 70-79.

TRAIDEX Needs and Implementation Study. (Final Report) DARPA Contract #MDA903-75-C-0224 by SofTech, Inc., May 1976.

Tully, Colin J. "Toward a Conceptual Framework for Systems Methodologies." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 609-612.

Udwadia, F. E. "Organizational Management--A look Beyond the Rational View." Journal of Management in Engineering, Vol. 2, No. 1, Jan. 1985, pp. 57-68.

van Lamsweerde, Axel, Delcourt, Bruno, Delor, Emmanuelle, Schayes, Marie-Claire and Champagne, Robert . "Generic Lifecycle Support in the ALMA Environment." IEEE Transactions on Software Engineering, Vol. 14, No. 6, June 1988, pp. 720-741.

Vernon, M.K. and Estrin, G. "The UCLA Graph Model of Behavior: Support for Performance-Oriented Design." In Methodologies For Computer System Design. Ed. Wolfgang K. Giloi and Bruce D. Shriver. Amsterdam:North-Holland, 1985, pp. 47-66.

Verity, John W. "Is It Real or Is It Information?" Datamation, May 15, 1986, p. 23.

Vessey, Iris and Weber, Ron. "Research on Structured Programming: An Empiricist's Evaluation." In Human Factors in Software Development. Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 474-484.

Vitarli, Nicholas P. and Dickson, Gary W. "Problem Solving for Effective Systems Analysis: An Experimental Exploration." In Human Factors in Software Development . Ed. Bill Curtis. IEEE Computer Society Order Number 577. Los Angeles: IEEE Computer Society Press, 1985, pp. 252-260.

Von Staa, A. and da Rocha, A. R. C. "Towards a Model For Evaluating Specifications." In System Description Methodologies. Eds. D. Teichroew and G. David. Amsterdam: North-Holland, 1985, pp. 295-311.

Walker, Theodore D. Perception and Environmental Design. West Lafayette, Indiana: PDA Publishers, 1971.

Wasley, Robert S. "Learning to Live Together." The Australian Computer Journal, Vol. 7, No. 3, Nov. 1975, pp. 105-108.

Wasserman, A. I. "The User Software-Engineering Methodology: An Overview." Proc. IFIP Working Group 8.1 Conf. Comparative Reviews of Information Systems Design Methodologies. IFIP, Geneva, 1982, pp. 591-628.

Weinberg, Gerald M. and Freedman, Daniel P. "Reviews, Walkthroughs and Inspections." IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, January 1984, pp. 68-72.

Westermeier, J. T. and Millstein, Julian S. "Combined Hardware and Software ("Turnkey") Contracts." In Computer Contracts and Current Issues, 8th Annual New England Computer Law Conference 1988. Sponsored by Massachusetts Continuing Legal Education, Inc. Boston: Bateman and Slade, 1988, pp. 57-164.

Weiss, David M. "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility." In Tutorial on Models and Metrics for Software Management and Engineering. Ed. V. R. Basili. New York: IEEE Computer Society, 1980.

Weizenbaum, Joseph. Computer Power and Human Reason. New York: W.H. Freeman and Company, 1976.

Westin, Alan F. Information Technology in a Democracy. Cambridge: Harvard University Press, 1971.

White, John A. "Automation: A New Perspective." Modern Materials Handling. Jan. 1987, p. 25.

White, Stephanie. "Requirements Modeling for Embedded Systems." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 238-240.

Wilding, John M. Perception, From Sense to Object. New York: St. Martin's Press, 1983.

Wing, Jeannette M. "Specification Firms: A Vision for the Future." In Third International Workshop on Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 1985, pp. 241-243.

Young, R.M. and Harris, J. E. "A Viewdata-Structure Editor Designed Around a Task/Action Mapping." In People and Computers: Designing For Usability. Cambridge: Cambridge University Press, 1986, pp. 435-446.

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

Zeimetz, Jordene. "Professional Viewpoint." PC Tech Journal, Vol. 6 No. 8, August 1988, p. 160.

Zinman, Marion J. "Computer Arbitration: The Program of the Future." Rutgers Computer & Technology Law Journal, Vol. 10, No. 1, 1983, pp. 103-108.