JPL Publication 88-41

Incremental Development and Prototyping in Current Laboratory Software Development Projects: Preliminary Analysis

Martha Ann Griesel

December 15, 1988

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. 88-41	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Incremental Development and Prototyping in Current Laboratory Software Development Projects: Preliminary Analysis		5. Report Date December 15, 1988 6. Performing Organization Code
7. Author(s) Martha Ann Griesel		8. Performing Organization Report No.
9. Performing Organization Name and Address JET PROPULSION LABORATORY California Institute of Technology 4800 Oak Grove Drive Pasadena, California 91109		10. Work Unit No. 11. Contract or Grant No. NAS7-918 13. Type of Report and Period Covered JPL PUblication
12. Sponsoring Agency Name and Address NATIONAL AERONAUTICS AND SPACE ADMINISTRATION Washington, D.C. 20546		14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

Several Laboratory software development projects that followed nonstandard development processes, which were hybrids of incremental development and prototyping, are being studied. In this report, factors in the project environment leading to the decision to use a nonstandard development process and affecting its success are analyzed. A simple characterization of project environment based on this analysis is proposed, together with software development approaches which have been found effective for each category. These approaches include both documentation and review requirements.

17. Key Words (Selected by Author(s))	18. Distribution Statement
Computer Programming and Software	Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified		39	

JPL Publication 88-41

Incremental Development and Prototyping in Current Laboratory Software Development Projects: Preliminary Analysis

Martha Ann Griesel

December 15, 1988

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

ABSTRACT

Several Laboratory software development projects that followed nonstandard development processes, which were hybrids of incremental development and prototyping, are being studied. In this report, factors in the project environment leading to the decision to use a nonstandard development process and affecting its success are analyzed. A simple characterization of project environments based on this analysis is proposed, together with software development approaches which have been found effective for each category. These approaches include both documentation and review requirements.

CONTENTS

1. INTRODUCTION 1
   1.1 SOFTWARE ENGINEERING BACKGROUND 1
   1.2 RESEARCH METHOD 3
   1.3 REPORT STRUCTURE 4

2. CASE STUDIES 5
   2.1 CASE A 5
      2.1.1 Decision-Making Factors 6
      2.1.2 Development Process 7
      2.1.3 Retrospective 8
   2.2 CASE B 10
      2.2.1 Decision-Making Factors 11
      2.2.2 Development Process 12
      2.2.3 Retrospective 14
   2.3 CASE C 14
      2.3.1 Decision-Making Factors 15
      2.3.2 Development Process 16
      2.3.3 Retrospective 16
   2.4 CASE D 18
      2.4.1 Decision-Making Factors 18
      2.4.2 Development Process 19
      2.4.3 Retrospective 22
   2.5 CASE E 22
      2.5.1 Decision-Making Factors 23
      2.5.2 Development Process 23
      2.5.3 Retrospective 26

3. CONCLUSIONS AND RECOMMENDATIONS 27
   3.1 CHARACTERIZATION OF PROJECT ENVIRONMENTS 27
   3.2 SOFTWARE DEVELOPMENT PROCESS ATTRIBUTES 30
      3.2.1 Experimental Type 30
      3.2.2 Evolutionary Type 31
      3.2.3 Established Type 32
   3.3 PRELIMINARY CONCLUSIONS 32

REFERENCES 35

FIGURES

1. Typical System Life Cycle 2

2. Case A Software Development Process 9

3. Case B Software Development Process 13

4. The Prototyping Paradigm and its Relationship to the Conventional Software Development Paradigm 17

5. Case D Software Configuration Management Process 21

6. Case E Software Development Process 25

TABLES

1. Classification of Case Histories 29

SECTION 1

INTRODUCTION

The purpose of this study is to discern and document the decision points which have led to using incremental development and prototyping approaches in Jet Propulsion Laboratory (JPL) software development projects. Project histories will be analyzed to provide guidelines to managers for selecting prototyping and incremental development processes, and to identify key elements and characteristics of these processes. The focus of the initial phase of the work is on documenting, in case studies, the environment in which the decisions were made, the specific project factors that prompted each decision, and effects of the decisions on software development.

As the experience base of project histories grows, those aspects of project environment critical to software development can be identified and used to characterize the environment. Basic similarities in development processes then can be identified, abstracted from their specific project environments, and codified. This will provide developers and managers with software development process paradigms, including documentation and review requirements, to choose in place of the conventional development paradigm, and the criteria for choosing them.

This report presents case studies from five Laboratory projects, a preliminary characterization of project environments based on analysis of these cases, and some software development strategies they suggest.

1.1 SOFTWARE ENGINEERING BACKGROUND

The standard Laboratory system development process, as presented in [7], is illustrated in Figure 1. This complete life cycle spans system life from concept to retirement. The software development phases reflect the conventional paradigm for systems whose requirements can be reasonably well specified at project initiation, and for which the development environment is stable. The software development processes considered in this study, incremental development and prototyping, represent methods for managing risk in an uncertain and changing environment. Examples of such environments are evolving requirements and changing hardware technology. These are not new approaches to software development; they are used in practice and discussed frequently in the literature [8, 15, 16]. Nor do they fall outside the standard development process, which can be tailored for different environments. However, in uncertain environments, incremental development and prototyping approaches usually have been applied on an ad hoc basis. For these cases they lack accepted representative paradigms, the associated guidelines for tailoring the conventional development process, and criteria for when to use them.

Both the terms "incremental development" and "prototyping" are used in the literature with several meanings [8]. For the purpose of this study, they are defined as follows:

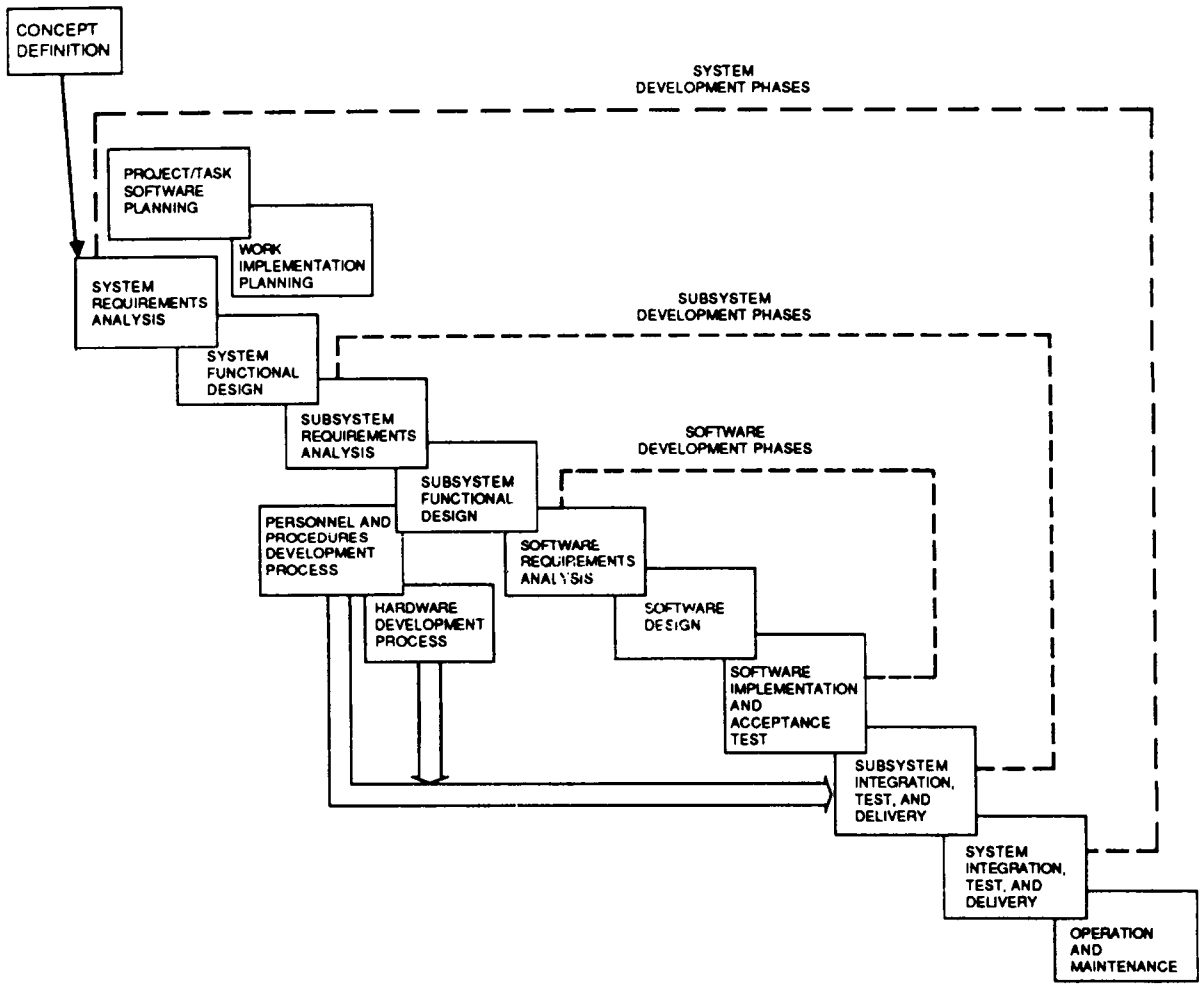


Figure 1. Typical System Life Cycle

From: JPL Software Management Standard [7], page 1-6.

Incremental development is a series of complete, robust implementations of functional subsets of system requirements.

A prototype is an implementation (of the entire system or a specific functional element) in which certain aspects of the implementation have been minimized in order that others may be maximized.

A typical prototype might minimize robustness and efficiency in order to maximize the number of included application functions and shorten the schedule (thus minimizing the risk of not having a baseline functionality delivered by an immovable deadline). Note that the definition of incremental development assumes that a complete set of system requirements exists, although they all need not be well-specified. The two concepts can simultaneously apply to the same project in two ways, both of which have been implemented in Laboratory projects.

(1) A prototype, where areas to be minimized have been identified, may have functional subsets implemented by increments; in this case "complete" has a reduced definition corresponding to how each functional element is to be prototyped.

(2) A project following either a conventional or an incremental development paradigm may prototype a functional element scheduled for later implementation in parallel with its main development.

Such mixed modes are being found more frequently than pure development of any type, suggesting that managers and developers are trying multiple approaches when faced with uncertainty. This phenomenon emphasizes the necessity of determining why a nonstandard development process was chosen, because different motivations are associated with different project environments. For example, some prototypes were developed for proof-of-concept, and some to help the sponsor become sufficiently familiar with automation to make requirements generation possible.

1.2 RESEARCH METHOD

The data collection procedure was adapted from standard research methods used for exploratory and descriptive studies [13 Chapter 3]. It includes both interviewing project personnel and studying pertinent project documents. Project documents for the case histories are given in the references. A discussion of the method is given in the research plan [2]. Information from the interviews and documents form the case studies which constitute the experience base for analysis. Hypotheses suggested by analysis of the experience base will be tested in future work. This will provide a basis for understanding the characteristics and structure of both the software development environments and processes, and the relationships between them.

1.3 REPORT STRUCTURE

Case studies for several Laboratory software development projects will be given in the next section. Each study includes:

(1) A brief description of the system to be developed and the sponsor environment.

(2) A discussion of decisions which led to using a nonstandard development process.

(3) A schematic of the development process, or one of its key elements, showing the associated documentation and reviews.

(4) Appraisal by the individuals involved of what worked and what did not, identifying the advantages and pitfalls they see in retrospect.

Factors contributing to decisions were identified by project personnel analyzing what had occurred. These factors were not always consciously recognized at the time the decisions were made.

The report concludes with a discussion of those points that arise in enough cases to start abstracting information, structure, and decision-making guidelines, and with a proposed project environment characterization.

SECTION 2

CASE STUDIES

Two general factors were found in all cases, and arose in almost all interviews. These will be presented first, and not repeated in every case.

(1) When creating their software development plan, individuals relied heavily on their personal experience. Those who had previously worked on successful projects that followed a conventional paradigm were proponents of these more traditional methods; those who had worked on successful prototypes were sensitive to opportunities to develop prototypes. Conversely, individuals who, directly or vicariously, had had a previously unsuccessful experience with a given software development process did not want to use that process. This was particularly true with prototyping.

(2) Schedules rarely permitted using resources to evaluate available tools, or learn to use them. Therefore those tools familiar to the developers, and already procured, were usually chosen; a recurring example was screen generators. In some instances, perceived lack of evaluation, learning, and procurement time resulted in few tools being used, even when the developer thought they would be helpful.

One observation also applies to the set of projects, rather than to any one particular effort. Those projects that were most successful completed each prototype, or increment, in about the planned time. An excessive amount of pushing planned implementation segments off to a later increment, or totally redoing a prototype, was usually an indication of trouble. Such delays often meant that requirements evolution was out of control, or that communications among sponsor, user, and developer were too noisy.

One note on terminology: the term "project" is used throughout these case studies to denote a software development activity with a single purpose, sponsor, and management structure. It does not necessarily denote a Laboratory Project; some case studies are of Projects, some are Division Managed Tasks, and others are single activities within larger NASA Programs.

2.1 CASE A

This case illustrates a proof-of-concept prototype that became an operational system. This prototype was requested by a sponsor who had previous experience with software development at the Laboratory and needed rapid system development. In return for cost and schedule considerations (project was to be completed within 24 months at fixed total cost), the sponsor waived their extensive software documentation and review requirements, and requested that the Laboratory do likewise. They accepted the risk that

system development on such a short time schedule might prove infeasible, and that the system would either never be completed or would fail. The sponsor also provided hardware very early in the project, both at JPL and at their own operational sites, and wanted the on-site hardware operating, even on a very minimal system, as soon as possible. An in-place manual system was being automated, and most of the users were not computer literate. The system was to perform near-real-time monitoring of resources (e.g., transportation flights) simultaneously at several geographically dispersed locations. This involved the development and management of the following:

(1) Local area networks and a wide area network.
(2) Distributed mini/super-mini computer system.
(3) Replicated, survivable, synchronized databases.
(4) Ultra-large screen display systems.

Work was performed under the usual Laboratory "best effort" agreement.

2.1.1 Decision-Making Factors

Lack of sponsor's computer literacy - Although there were a few individuals in the sponsoring organization who had tried to automate some of their existing manual system on a personal computer, most of the users were not computer literate. It was hard for them to determine how to implement their requirements because they did not know the potential of an automated system. In addition, the Laboratory was not familiar with the problem domain. The first prototype was therefore developed to give the sponsor something, however minimal functionally, to use and gain experience, and give Laboratory personnel a chance, through interaction with the users, to learn the problem domain. This initial prototype was intended to be thrown away, but became the first increment for an operational system.

Lack of existing high-tech system - A small increment of the system functionality, even five percent, was of use to the sponsor, especially because the first increment would include the basic network, providing a needed communications capability. Hence, an incremental approach gave the sponsor a growing, useful system as expeditiously as possible. This approach would not necessarily work if an existing system were being replaced with a "higher tech" system.

Sponsor wanted in-place hardware put to use as soon as possible - Using off-the-shelf software wherever possible accelerated initial operational capability. For example, the first increment included a few applications functions, a local area net, and an off-the-shelf mail package. In addition, the architecture from another Laboratory project that faced similar networking and distributed system problems was reused, saving both time and uncertainty in design. This was possible because of the serendipity factor in effect when individuals move from a shrinking to a growing project, bringing ideas with them. The technical focus of line organi-

zations in a matrix management structure makes this more likely, because ideas from past projects are more apt to be relevant.

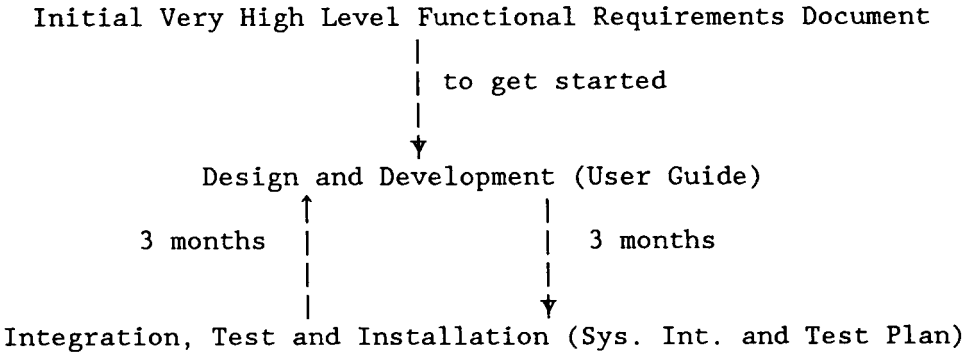
Need to keep sponsor motivated - Several aspects of the close, cooperative relationship established with the sponsor coupled sponsor satisfaction and the software development methodology.

> Requirements team - This comprised both sponsor and Laboratory individuals, the expert users of the system helping to write the requirements. They started with a high-level "wish list" and kept tailoring it throughout the life of the project as both user and developer became more familiar with both the problem and the opportunities for automation. This led to increments that were vertical (new depth to an old functionality) as well as horizontal (new functionality).
>
> Sponsor participation in testing - System development benefitted from functional validation of each increment by the expert users uncovering some of the problems. This helped minimize Laboratory personnel time in some of the test phases, thus helping to meet the short development schedule.
>
> Releases every six months - Frequent software releases kept the vital feedback channels open with the users (they had something new to look at), and kept the sponsor confident the schedule was being met.

2.1.2 Development Process

A modified form of incremental development was used because not all the requirements could be determined until the sponsor had some experience with the system. Thus increments added both more functionality in already developed areas and new functionality. Also, after the first release, system and function development was simultaneous. Adherence to the sponsor's extensive documentation and development cycle was waived in interest of cost and schedule since the system was a prototype. However, the prototype quickly evolved into an operational system, and the project was asked to document each release "after the fact." This was successful for two reasons: first, the development team was sufficiently integrated that architecure and module interface issues could be handled in frequent design meetings; and, second, the design of workstation screens, which was a large portion of the design effort, was done by successive screen prototypes iterated with the user and concurrently documented in User Guides. These User Guides became surrogate design documents, with the design presented from the user's point of view. Final system design documentation was written specifically to satisfy system maintenance requirements.

Schematically, the six-month development cycles were as follows. Documents associated with each cycle segment are given in parentheses.



A more comprehensive illustration of the software development process is given in Figure 2. The User Guides became the requirements documents for the implementation of each increment. A single Design Book, including data flow diagrams and source code, was maintained on the computer with the system under configuration control. Thus, at any time there was only one copy. This Design Book was to be delivered with each release. Note that each phase of an increment has its documentation: the User Guide for design and development, and the System Integration and Test Plan (SITP) for integration, test, and installation. More formal design documentation required for maintenance was to be delivered with the final system.

Referring to Figure 2, note the key role of the Configuration Control Board in damping requirements and design evolution. Once an increment had been delivered, the individual user could no longer ask the individual implementor for a change, as was the normal mode of operation during the six-month development period. Changes had to be submitted to the board and added to the "wish list." Only those approved by the board were implemented. The damping action provided by the Board was reflected by the rate of growth in the number of approved change requests: 70 approved requests during the first one-third of the project, 100 during the first half, with none during the second half.

2.1.3 Retrospective

(1) Close cooperation between sponsor and developer is crucial when a system is being evolved, not built to predetermined specifications, on a relatively inflexible schedule. This cooperation extends to both technical areas, e.g., working together to evolve requirements and to test the system, and administrative areas, e.g., providing development hardware when needed.

(2) This approach to incremental development, using six-month increments and adding functionality both vertically and horizontally with User Guides as surrogate design documents, should work well on horizontal, screen-driven (i.e., highly interactive) applications. As used here, "horizontal" means a system comprising several loosely coupled applications subsystems.

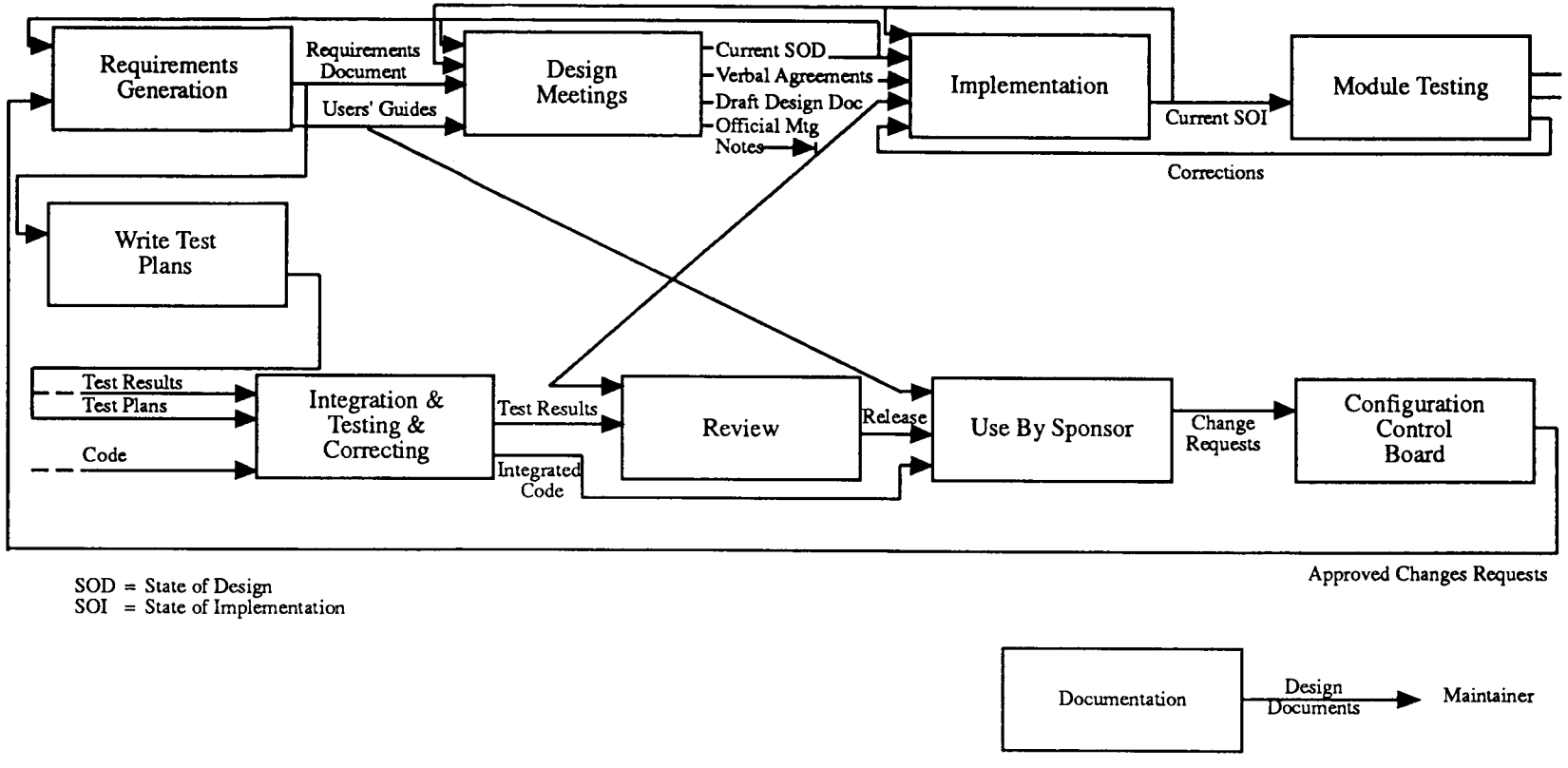


Figure 2. Case A Software Development Process

(3) The Laboratory set up one-to-one points of contact with key individuals in the sponsor's organization, the expert users functionally responsible for an area. These key sponsor personnel were specifically assigned to work with the developer, and kept in position, and out of normal job rotation, until the completion of the system. This mirroring of the sponsor's organization supported good technical communications and cooperation throughout the project.

(4) The hardware was decided on and made available shortly after the beginning of the project and supported the off-the-shelf software that was needed for rapid development.

(5) The informal Design Book was not found to provide sufficient documentation for maintenance. A specific effort to analyze and document the design of the delivered system was required.

(6) The moderate size (about sixty people) and reasonably restricted technical focus of this project allowed the close teamwork essential for a more loosely structured, less formal development process. Most of the people were very experienced software developers. This approach to system development may not work with a large number of novices on the team.

(7) A configuration control board was used to stop requirements evolution (see Figure 2). Final requirements, agreed upon by the board, are necessary for final system delivery.

2.2 CASE B

This case illustrates the direct translation of a manual system, requested by the sponsor as a proof-of-concept, and illustrates the use of prototyping in proof-of-concept development. Specifically, the sponsor wanted to determine if automation in the form of microcomputers operating in a local area network environment would help in resource management for a large, complex airlift unit. Resources included aircraft, personnel, material handling, maintenance and support equipment, and supplies. The lessons learned from the prototyping process were to be used by the sponsor in drafting the Request for Proposal (RFP) for the operational system. Experience with the prototype would also raise the level of computer literacy of the sponsor's user community (from near zero), further supporting the RFP.

The system included five functional area nodes, each requiring a local area network. Communications was also required between these nodes. After the preliminary design phase it became evident that the choice of hardware was constrained. The sponsor wanted the individual workstations all to be the same personal computer specified by the sponsor. Each workstation was also to provide, besides its functional area capability, standard office automation. In this prototype, requirements gathering and design were minimized to reduce schedule and cost.

2.2.1 Decision-Making Factors

Requirements gathering was to be minimized - System requirements were to emulate the manual system that was already in place. Existing procedures were also tied to the manual system, and could not be considered for change until the capabilities of an automated system were demonstrated.

Off-the-shelf tools were used - An evaluation of available tools was made and the best tools procured. Available tools were very poor.

Design phase minimized - This prohibited initial development of standard tools packages. Thus, some functional elements implemented later in the development, after some tools had been developed in parallel with functionality, were faster and more user friendly than those implemented earlier.

Inconsistency in implementation of the manual system by different personnel - Several factors in this area contributed to making a stable design difficult. This is instability of requirements, not their healthy evolution. As an example, the man-machine interface was different in different areas.

> The individuals performing the tasks being emulated were periodically transferred, and although the tasks and procedures were the same, each individual's approach to a task was slightly different.
>
> Interacting with the developer was an additional duty for sponsor personnel, giving it less priority and time. Contrast this with Case A, where supporting the developer was a recognized part of the user's job.
>
> The initial Functional Requirements Documents, which had been agreed to by the sponsor, could only generally convey what was desired in the automated system. This was due partly to the sponsor's lack of computer literacy, and consequent lack of understanding of the potential of an automated information system.

The sponsor's inability to send personnel to the Laboratory to get early hands-on experience with the development system also may have slowed design stabilization.

Use of off-the-shelf technology - The sponsor directed that off-the-shelf hardware and software be used whenever possible. This led to greater risk when the workstations were constrained to be specific personal computers (based on criteria other than the availability of commercial software), because the choices of compatible hardware and software were thereby limited. Products which performed the required functions on the specified hardware were not always available from established vendors. Dealing with newer vendors increased the risk of not meeting the development schedule (because the vendor's promises might be over optimistic) and subsequent maintenance (because the vendor might disappear).

Only one set of development hardware - This required test of the current increment simultaneously with the initial development of the next increment, which affected increment development schedules.

2.2.2 Development Process

The project followed traditional methods to the extent possible, taking into account that this was a proof-of-concept in support of an RFP, and required rapid development. The project was conducted in phases, as follows. A more detailed diagram of the overall development process is given in Figure 3.

System Definition Phase - initial planning and conceptual design. Documents generated were: Functional Requirements (FRD), Functional Design (FDD), RFP for development hardware and associated software (including networking). The development process elements, with associated documentation, were as follows. This corresponds to the portion of Figure 3 from requirements analysis through hardware selection. Associated documents are given in parentheses.

```
General Requirements ----> Issue RFP ---> Redesign for HW
(FRD, FDD, RFP)                           (Revise FDD)
```

Implementation Phase - consisted of three sub-phases.

(1) Architectural Design Phase - resulted in publication of General Design Document (GDD), based on the FDD. Also, the first version of some of the User Guides (bottom-up software design references) was developed, along with a preliminary version of the Programmer Reference Manual. Fundamental code was also developed and delivered at this time, including office automation. The development process elements were as follows.

```
Architecture and System Design --> Implement --> Test
(GDD, Prelim. User Guides
 and Programmers' Ref. Manuals)
```

(2) Phase One - foundation, communications, database software.

(3) Phase Two - baseline set of database display and edit applications, and communications software.

(4) Phase Three - included both enhancements to Phases One and Two software (vertical increments) and additional functionality (horizontal increments).

Each increment was developed following a development process based on a standard paradigm. This is illustrated in the system design through delivery segment of Figure 3. System test and integration was hampered, or new increment design was hampered, by the necessity of performing both on the same hardware. Neither formal acceptance testing nor code audits were

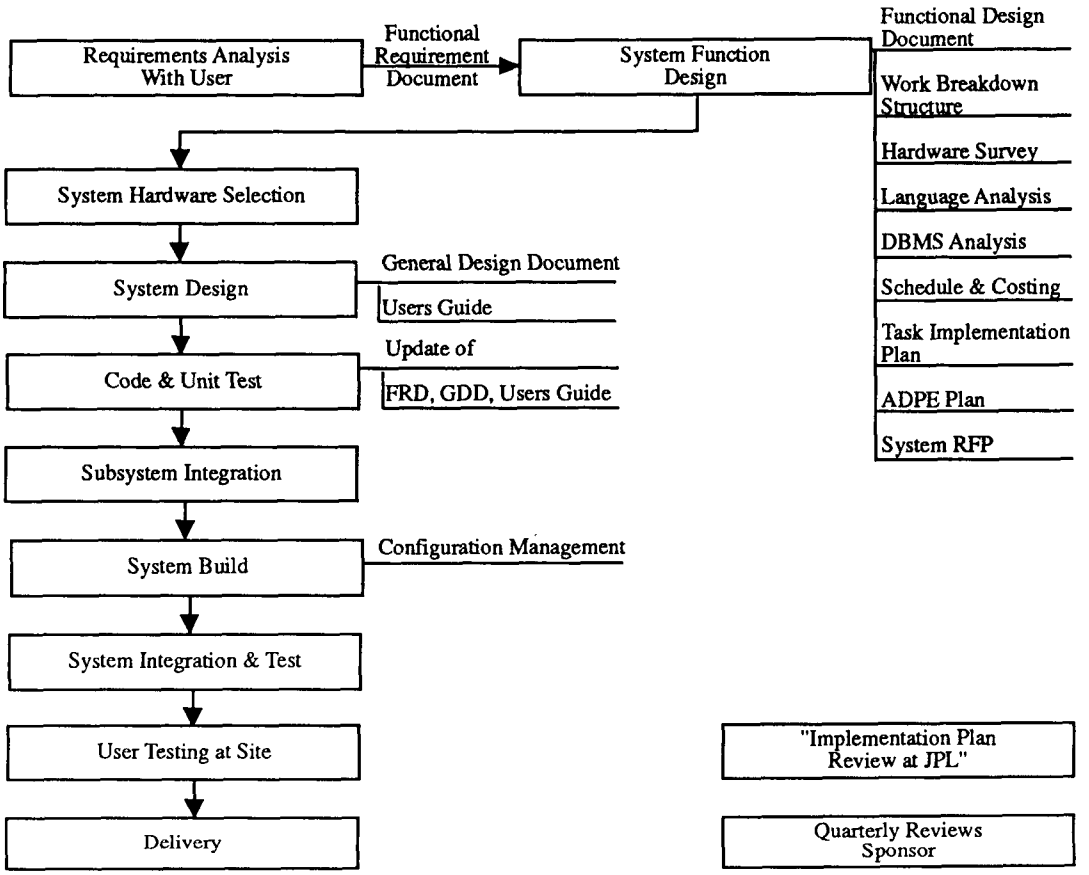


Figure 3. Case B Software Development Process

performed due to lack of funding, the timing of software deliveries, and the nature of the system being developed, i.e., not an operational system. Each of the final increments was developed in three to six months. A "Lessons Learned" document was also produced both to support the RFP and subsequent system development, and to prevent the rediscovery of pitfalls.

2.2.3 Retrospective

(1) Both development and integration and test hardware are needed for rapid incremental development. Being able to begin development on the next increment while finishing integration testing on the current one can save schedule time. This was also shown in Case A.

(2) Even with a minimal set of documentation, an interface control document under configuration control is needed.

(3) Obtaining rapid development by minimizing the design phase led to five different screen generators where one generic one would have been better. These were mainly off-the-shelf. More time was needed for the normal activity of analyzing and integrating the requirements found by different requirements gatherers who talked to different users and covered different functionalities.

(4) Design phases lacked the necessary face-to-face interaction with the users. The continual personnel changes in the sponsor's organization added to this problem. Contrast with the experience in Case A where the sponsor was able to keep personnel in place throughout development.

(5) Using commercial off-the-shelf software from an as-yet-unproven vendor can introduce risk in meeting schedules dependent on vendor performance, and assuring system maintenance. Availability of commercial software should be taken into account when choosing hardware whenever using off-the-shelf software is deemed otherwise advantageous.

(6) A "Lessons Learned" document is very valuable. It helps preserve not only the understanding of good ways to inject automation into a problem domain, but also the knowledge of where potential problems and blind alleys lie.

2.3 CASE C

This case illustrates prototyping in parallel with and supporting a large software development project which is following a conventional paradigm. It arises from an attempt to minimize the risk associated with change for the large ground information system that performs data capture and processing of engineering and science data for planetary and earth observing missions. Changing requirements from new instruments and new missions, and the enhanced capability made possible by new technology, result in

system changes. Prototyping a new technique, or configuration of hardware and off-the-shelf and JPL developed software, in parallel with and a little in advance of system software development can provide the following to the developer:

(1) Insight into what is "safe" to incorporate in the main system.

(2) Analysis of the relative merits of different technologies and configurations.

(3) Strategies for integration into the main system.

The prototype itself is not intended to be integrated into the system. Technology is transferred chiefly by transferring the individuals who developed the prototype into main system development.

2.3.1 Decision-Making Factors

This case concerns only the prototyping activity and its relationship to main system development. It differs from the other cases in that it covers many small activities, e.g., a prototype of an individual functional element or communications architecture, and not the development of a single integrated system. The ground information system which this prototyping effort supports is developed using a modified conventional paradigm which includes some incremental development, often of large, complex increments. This allows smoother insertion of the results of the prototyping activity.

Needed to determine if off-the-shelf software was acceptable - Traditionally, software had been developed specifically for this application; off-the-shelf packages were believed to be neither sufficiently robust nor sufficiently flexible to handle the specific problems encountered in such a large information-handling system. Commercial packages were built into small prototype systems to determine if they could meet specific application needs. Note this is not the same as testing a commercial product in an applications vacuum.

Needed to change data base hardware/software - The only prototyping in this instance was the testing to determine how well various available products performed in specific real-time environments. Such experimentation could not be done with the entire system, but needed a separate, smaller parallel capability.

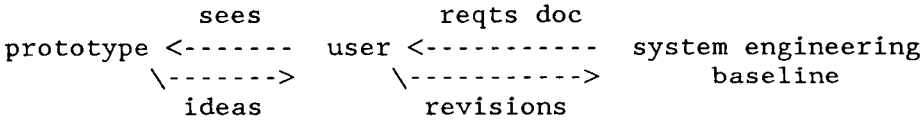
Desired near-real-time capability for low rate science instruments - Currently, images produced by some instruments on a platform are processed in near-real-time during an encounter, but science users of other instruments have to wait for more fully processed information. A system is being developed to perform near-real-time processing for some of these other instruments. Rapid incremental development was chosen in order that the capability be available during the next encounter. The system is a prototype in the sense that providing science functionality on time and within

budget is maximized, with the understanding that some fatal error could occur during encounter, thus minimizing some robustness, error handling, testing, etc. Note this is basically the same risk assumed in Case A, that the system may not work.

2.3.2 Development Process

These prototype activities functioned in parallel with the traditional ground information system design effort. General methods for incorporating prototyping in the conventional software development paradigm have been discussed in the literature [8]. Figure 4 illustrates the general relationship.

Technology transfer was often through the individual user who interacted with both the prototype and the related portion of the ground data system. As illustrated below, the system user sees and understands the potential of the prototype and incorporates it in the main system requirements document.



Looking at Figure 4, the above mechanism is seen to correspond to the "extract information about user requirements" and "extract information useful to system design" paths. Individual engineers who worked on both the prototype and the related portion of the ground information system provided another important means of this technology transfer that links the two development processes.

Documents were used to save and distribute information. Those used in the activity were:

(1) Functional Requirements Document.
(2) Design memoranda.
(3) Interim reports.

Demonstrations were used extensively. Due to the role this prototype activity played in absorbing the risk and doing proof-of-concept for the main system, the demonstrations were considered more important than the documents.

2.3.3 Retrospective

(1) This case points out the experimental aspect of prototyping. Reducing the risk associated with introducing new technology into on-going systems, or system upgrades, requires learning by experimenting with the new technology in the context of the application. In this case,

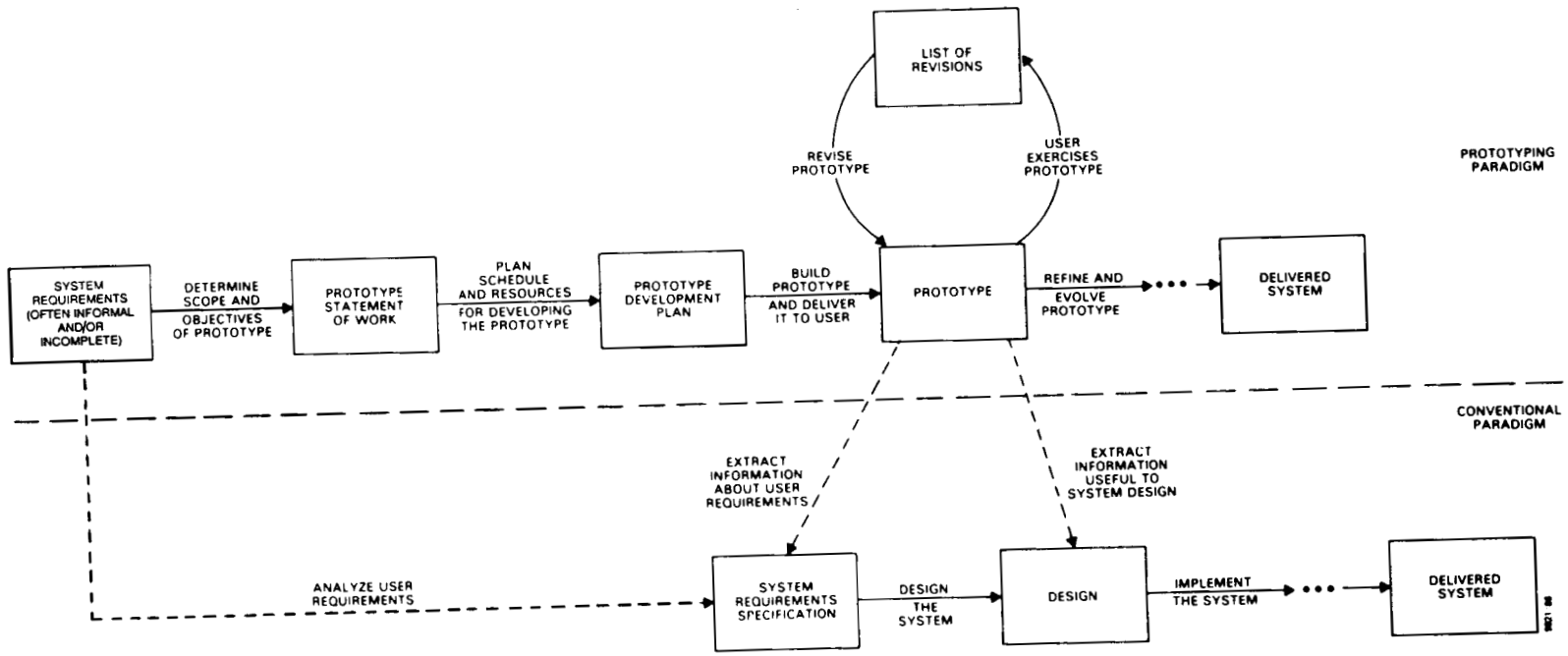


Figure 4. The Prototyping Paradigm and its Relationship to the Conventional Software Development Paradigm

From: Agresti, New Paradigms for Software Development [8], p. 7.

prototyping has proven to be a method for experimenting for reasonable cost.

(2) The real technology transfer occurred when the individual who developed the prototype returned to the main system development project and took the ideas and experience and vision gained in his head - not in documents or in software. This is a typical technology transfer mechanism, and works well for such "local" transfers.

2.4 CASE D

This case illustrates software development of a system whose heart is a simulation model. The sponsor wanted to simulate complex outcomes of decisions by competing decision makers. The resulting system was to be used to train large groups of decision makers and their supporting staffs. Simulation results were shown on workstations. These outcomes were to seem "real" to the trainees using the workstations; for the sponsor's application, this required special graphics. The training system was to be data-driven, in order that the trainers and trainees could change the scenario. A minicomputer was required to process the resulting complex simulation. Several microcomputers were used to control the workstations. These processors were locally connected by local area networks and globally by a wide area network. The simulation was written in Simscript, with all environment and process descriptors contained in a database.

System development included both prototyping and incremental approaches. At the beginning of the project, the sponsor was only partly computer literate, and the Laboratory was not applications literate. Also, the Laboratory was only slightly familiar with the simulation language. Often, before an applications function requested by the sponsor could be understood and modeled by Laboratory personnel, consultation with a domain expert was required. These domain experts frequently did not work directly for the sponsor, but in another part of the sponsor's organization. This three-way interaction further complicated the learning process. The sponsor requested use of the most current version of the system (even if not yet formally delivered) for periodic training. This interaction influenced the functionality requested in the next increment, and, as in Case A, provided some system testing. The project was conducted under a standard "level of effort" agreement.

2.4.1 Decision-Making Factors

Requirements understood by neither the developer nor the sponsor - The sponsor had only general goals that translated into quite general requirements. Furthermore, the Laboratory's initial unfamiliarity with the sponsor's problem domain hampered the developer's capability to contribute to the initial requirements. Thus, the requirements evolved as the sponsor saw successive versions of the system and Laboratory personnel gained understanding of the problem domain. The initial system was a

learning experience with respect to subsequent system development, as is the current operational version with respect to the next major version.

Sponsor's desire to use undelivered "most current" versions for training exercises - Training exercises were treated as functional validations. Each new increment was usually fairly thoroughly tested in such an exercise by the time it reached the final integration test phase. Each small piece already had completed integration and test as it was incorporated in the system. Some of these exercises involved many trainees, making the functional validation reasonably thorough. This degree of thoroughness was not possible on the smaller development system which had, for example, fewer workstations. One hundred requests reflecting software problems and requirements modification desired by the sponsor were sometimes generated at such a functional validation.

Sponsor requirements were evolving - Repeated revisions of requirements can lead to extensive rework, causing schedule slips and budget overruns. By developing small enough increments, excessive rework was often avoided. Incremental development can be used in this manner to reduce the risk associated with evolving requirements.

Sponsor maximized some factors at expense of others in each increment - Even operational versions were prototypes in the sense of this study. For example, one version maximizes speed of system operation and distributed system aspects.

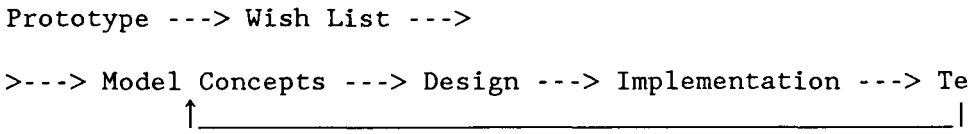
2.4.2 Development Process

Three systems are being sequentially developed, two of them operational. However, each of the first two will have been a prototype to the third in two senses.

(1) Portions, both of application functions and system features (e.g., user interface), were minimized to maximize other portions at the sponsor's request. Only for the third system is the sponsor able to state complete system requirements at the beginning of development.

(2) Each successive system was sufficiently different from the previous one that only personnel, ideas, and some software tools were inherited from one to the next. Code initially inherited was usually radically modified, modules being rewritten two to seven times.

By the definitions being used for this study, the first version is considered a prototype, followed by incremental development of the next version. Thus, the development sequence is as follows.



The focus of this discussion will be the incremental development portion of the process, model through test. Cycles were 3 to 12 months in duration depending on the complexity of the functionality being added. Several functionalities, each following its own cycle, were added simultaneously. Internal builds, i.e., fixing the current state of the system, were frequent. Official deliveries to the sponsor were at approximately one-year intervals and contained an integrated set of functions. Not-formally-delivered code used for functional validation was not left in the sponsor's possession after the training exercises.

The following documentation was part of this development process.

(1) Sponsor's Wish List, i.e., requirements, in priority order.
(2) Model Concept (Software Requirements) Document.
(3) Model Software Design (for each concept).
    (a) Software modules.
    (b) Data structure.
    (c) Test plans.
(4) Design Documents.
    (a) High level programmers' maintenance guide.
    (b) Detailed design in pseudo code.
    (c) Workstation detailed design.

Code walkthroughs by a board consisting of the code author, modeler/designer, the software supervisor, and a few programmers selected by him, were conducted during implementation. This was believed to be essential for a large (250K lines of simulation language code), complex system under rapid development. It provided both a check on the code and functionality, and information exchange among development personnel.

Once a baseline system was in place, configuration management (Figure 5) became the heart of this highly adaptable process. In fact, Figure 5 actually shows the complete iterative software development process, beginning with the "sponsor" box at the top of the diagram. Engineering change requests for functional changes and test incident reports for problems found during integration test and functional validation were the formal vehicles for initiating change. Engineering change requests were also used to introduce functionality for a new increment. The proper concept and design reviews for implementing these changes were assured by the configuration control board, which met three times a week. Control of actual software and documents, which were changed concurrently with the software, resided with the librarian. Configuration control was maintained using commercial software. All software additions and modifications could be traced to formal change requests.

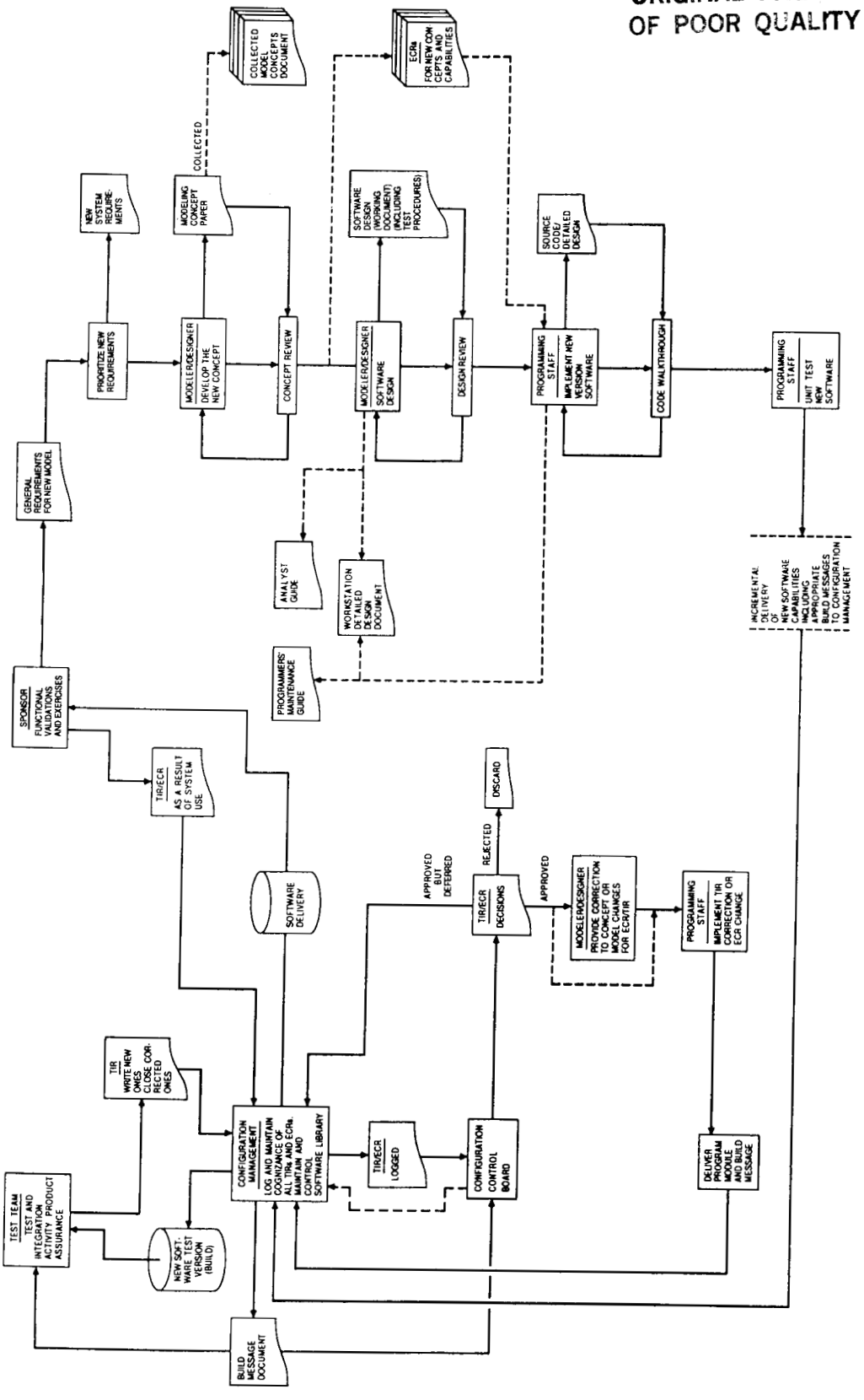


Figure 5. Case D Software Configuration Management Process

From: Project Software Management Plan

2.4.3 Retrospective

(1) Due to the way it grew from continually changing requirements, directly from concept to code, the system is not modular. Adding new capabilities can trigger hitherto hidden errors. In particular, at the beginning of the project many design decisions were made at the coding level, leaving a legacy of unforeseen global effects. Thus, the system has become difficult to fix or modify, potentially adding to maintenance costs.

(2) For this size software development effort, the configuration management system can be designed to effectively manage the iterations of the incremental process. This provides a mechanism for controlling and adapting to changing requirements.

(3) Code inspections (walkthroughs) are vital in this development environment. They should cover code readability, content, and structure: does it do what it is supposed to without obvious software errors, logic errors, or undesirable side effects; does it conform to project software standards; and is the documentation readable?

(4) The excellence of the system in meeting sponsor functional requirements is due to a great extent to the small group of very experienced, motivated modelers and programmers. Note the contrast with Case A in which experienced personnel were also relied on heavily, but software developers, not modelers.

(5) A cooperative sponsor, e.g., in providing access to application domain experts outside his immediate jurisdiction and to needed facilities and personnel for functional validation, makes this less structured, more evolutionary method of system development possible.

(6) The practice of using the most current system build, not-formally-delivered, for sponsor training exercises was valuable. It supported evolution of the requirements, increased both sponsor's and developer's understanding of the potential of automation in the application domain, and significantly enhanced the thoroughness of system functional testing. Similar experience was found in Case A.

2.5 CASE E

This case illustrates large data management system development for science community users. The database was designed to give scientists at diverse locations, usually universities, better access to data collected during NASA missions, both past and upcoming. This entailed the system performing four functions:

(1) Data quality assurance and standardization.
(2) Data cataloging.
(3) Data archiving.
(4) Data distribution.

A central node contains the high level catalog and distributed discipline nodes (one per discipline) the actual data, detailed catalogs and science data analysis tools. The system provides a data standardizing interface between mission design teams and university Principal Investigators (PIs). The project has been sponsored by NASA, and was not requested by the science community. PIs at universities often were restoring and cataloging their own subset of the data. A prototype was needed to demonstrate the value to them of a central catalog, data standards, and access among discipline nodes and universities. Incorporating software already completed or being developed at discipline nodes entailed imposition of some software development standards on the nodes with which they were not accustomed.

2.5.1 Decision-Making Factors

Need to demonstrate value of system to science community - An initial proof-of-concept prototype system was developed to demonstrate the following to the community of PIs:

(1) A central catalog node would benefit them.

(2) A more integrated and standardized system was better than continuing to "go it on their own."

The PI's active involvement was required for the system to support science needs.

Rapid changes in the technology needed for implementation - Hardware, which had performed satisfactorily, was in place at the completion of the prototype. However, the technologies involved, such as data storage devices and workstation displays, are continually changing. By the end of a several year development, these changes may have provided significantly superior hardware for the application. Thus, a multi-phase incremental type of development was initiated, allowing new technology to be considered and inserted at the beginning of each new phase.

Need to maintain funding - Both a second follow-on prototype and a pre-delivery (early delivery of partial functionality during integration and test) were scheduled to enhance active involvement and interest of the science community. These also aided in revising and solidifying user requirements.

2.5.2 Development Process

The project had two initial prototypes, and system development was in three increments.

prototype 1 ---> prototype 2 ---> increment 1 --->

>--->increment 2 ---> increment 3 = complete system

The initial prototype focussed on three separate aspects of system development:

(1) Developing a high level central catalog, with functionality limited to the cataloging function.

(2) Developing testbed discipline nodes at the PI's home sites, continuing the work by PIs already in progress on subsets of the data.

(3) Investigating the applicability of new technologies, such as optical disks and enhanced work station displays.

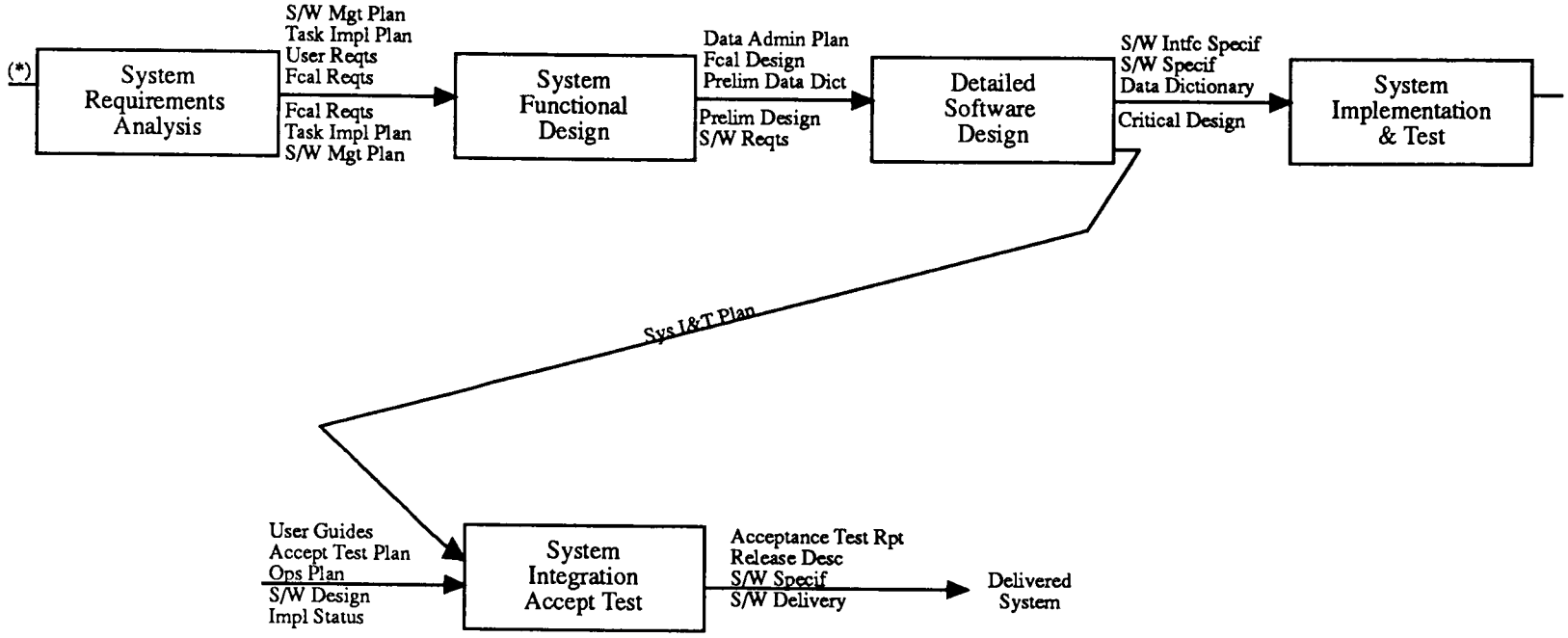
The second prototype of the catalog function integrated the central and a detailed catalog for one discipline and provided a two-thirds complete user interface, and data quality assurance and standardization. The demonstration of this prototype used real data.

The first increment inherited conceptual design (captured after the fact as was system design in Case A) and hardware from the first prototype. It inherited software tools, about half the implemented user interface, and most of the data quality assurance and standardization from the second prototype. Both prototypes helped define and modify user requirements. Software tools were also inherited from the participating portion of the scientific community. Existing networks were used to communicate between dispersed sites.

An increment, which is anticipated to take about two years to complete, is developed through a complete, conventional life-cycle, from requirements analysis through integration and test. The phases, a modified version of the JPL standard [7], are as follows.

(1) System Requirements Analysis.
(2) System Functional Design.
(3) Detailed Software Design.
(4) System Implementation and Test.
(5) System Integration / Acceptance Test.

This development process is illustrated in Figure 6. The first increment was not further subdivided into smaller increments to ensure a single consistent detailed design. This software development paradigm has been tailored to the specific needs of a system in which the integration of database technology is significant. The choice of documentation was also influenced by database structure and management considerations, e.g., with the addition of a Data Administration Plan. A User Requirements Document was also added to capture the requirements of the science community. Maintaining current documentation was important for both intra-project coordination and communication with the users.



(*) Documents specified above the line, Reviews below the line.

+ Based on Software Management Plan

Figure 6. Case E Software Development Process

2.5.3 Retrospective

(1) The first prototype was not documented, i.e., there was no written record of requirements, design, implementation (except the source code), testing or lessons learned. In reaction, the project began documenting subsequent work quite extensively. This has evoked two different responses:

   (a) The development finally has some form; time is no longer being wasted on divergent paths, and a viable system is beginning to emerge.

   (b) The documentation is in danger of becoming excessive for the size of the project. Completion of a document is so important that time is not always taken to ensure that the document is readable, or even that its content is complete and appropriate to the development phase (e.g., design) it represents.

(2) Turnover of personnel following demonstration of the first prototype exacerbated the loss of information due to lack of documentation.

(3) The prototypes were valuable in showing that the conceptual model was too ambitious and complex to be implemented within available development resources.

(4) Database standards (e.g., universal interface standards, labels, nomenclature) need to be in place early in the development cycle. However, these facets of the system are often not visible in early development, thus are often disregarded (see number 5 below).

(5) The developer will often focus on the visible parts of the system (e.g., user interface) when producing a prototype with a demonstration as the main review. A written prototype plan describing what is to be learned from the prototype is needed to ensure emphasis is properly placed during development.

(6) The incremental approach has led to some difficult negotiations with the science community. They fear that the project may terminate after the delivery of the first increment. Therefore, each discipline has tried to negotiate all its functionality into the first increment.

(7) Lack of formal change requests in the early part of development allowed creeping requirements without offsetting schedule adjustments. The user community has many requirements it wants satisfied, all immediately. Formal change control was found to be necessary to ensure the users understood the costs of their requested additions. This is analogous to the manner in which the configuration control board was used to control requirements evolution in Cases A and D.

(8) Definition of terms is even more crucial for a large data management system than a general software system. The data dictionary needs to include variable names as well as function and data nomenclature.

SECTION 3

CONCLUSIONS AND RECOMMENDATIONS

Choice of the software development process was found to depend primarily on two factors, with the first outweighing the second.

(1) Developers chose the process with which they had been most successful on previous projects.

(2) The project chose the process specifically requested by the sponsor, at least in name. However, agreement on a name does not mean agreement on a development process, because a name can be used for different processes, depending on the reference source.

Interviewees did not mention that any analysis had been performed in choosing a development process, although single factors, such as the need to quickly determine if system development was feasible, were sometimes cited. No specific guidance is available from current standards for analyzing or dealing with some of these more non-standard software development environments.

The case studies were analyzed to determine if a characterization of project environments could be developed on which the selection of a software development process paradigm could be based. The resulting characterization is given in the remainder of this section.

3.1 CHARACTERIZATION OF PROJECT ENVIRONMENTS

Three environmental attributes were isolated as influencing the applicability of a software development process paradigm. These attributes were discussed in all the interviews, often being introduced into the discussion by the interviewee.

(1) The degree of maturity of the end users' (and sponsor's) understanding of their requirements for an automated system, and the degree of precision with which the sponsor could state those requirements.

(2) The depth of the developer's (i.e., in this study the Laboratory's) understanding of the users' problem domain.

(3) The developer's level of familiarity with the hardware and software: had it been used in like projects in the past, or were new applications involved (e.g., hypercube or Ada)?

Software development projects were found to fall into three types that could be characterized by rating the project "low," "moderate," or "very high" for each of these attributes. Names will be given to the environment types for easier reference.

Experimental - The project was rated low for at least two of the attributes.

Evolutionary - The project was rated moderate for at least two of the attributes, and the user requirement attribute was never rated low.

Established - The project was rated very high for all three of the attributes.

Typical examples of the experimental type are:

(1) The initial stage of a project where the developer is trying to understand the problem and the user is trying to understand automation.

(2) The situation in which a well-understood system is being upgraded and is to be implemented on a concurrent processor, thus potentially affecting familiarity with both hardware and requirements.

Typical examples of the evolutionary type are:

(1) The evolving systems for which the user or developer gain insight into application of automation within the problem domain as experience is gained when each release is tested and becomes (in some sense) operational.

(2) The methodically growing system for which the requirements are known, but potential implementation opportunities and problems are unknown in a new hardware or software environment.

Typical examples of the established type at the Laboratory are the large ground information systems.

Table 1 gives the ratings and resulting environment types for the projects discussed in the last section. These ratings are based on the assessments made by different individuals on each project. Note that Case C is divided into two subcases. Case C_1 includes those activities for which the hardware or software being introduced was deemed to strongly affect requirements, resulting in a low requirements rating and experimental classification. In case C_2 are those activities with more incremental changes that did not extend as drastically into requirements, resulting in a moderate requirements rating. They were considered to be evolutionary increments of the main system.

Projects were found to progress through these types like stages, experimental to either evolutionary or established, evolutionary to established. An additional pattern was "experimental eddies" accompanying a conventional paradigm to determine if a requirement or technology change was appropriate. Examples of this were discussed in Case C. It is interesting to note that the use of prototyping in experimental and evolutionary type

Table 1. Classification of Case Histories

Case History:	A	B	C_1*	C_2	D_1^	D_2	E
Environmental Attributes							
Degree of Maturity of Sponsor's Requirements	L	L	L	M	L	M	L
Depth of Developer's Knowledge of Problem Domain	L	L	M	M	L	M	L
Level of Developer's Familiarity with HW and SW	H	M	L	M	L	H	M
Environment Type:	Ex	Ex	Ex	Ev	Ex	Ev	Ex

* C_1 and C_2 represent the different subcases of Case C

^ D_1 is the initial prototype, D_2 the subsequent development.

Environmental Attribute Ratings - low (L), moderate (M), very high (H)

Environment Type - Experimental (Ex), Evolutionary (Ev), Established (Est)

environments is very similar to that proposed in Mayhew and Dearnley's theory-based classification of prototyping [14].

The term "rapid" did not appear in any of the above. A rapid development may be of any type. Substituting a process suited for one type, for example, the experimental type, because it was perceived to lead to more rapid development than the paradigm for a type which better matched the project environment, was never found to succeed.

3.2 SOFTWARE DEVELOPMENT PROCESS ATTRIBUTES

Each project environment type led to different software development processes, including different normal lifetimes, documentation and review requirements, and demands on support from the project environment. In general, projects with less definition required a more exceptionally supportive environment. In this section, development process attributes will be given for each type. For the experimental and evolutionary types, the attributes come from analysis of the case histories. For the established type, attributes were drawn from discussions of the typical system life cycle given in the literature [8, 16].

3.2.1 Experimental Type

Most often used prototyping, with the goal being an initial statement of user requirements, architecture, and design concept.

Required a close, mutually supportive relationship between the user and the developer, with shared vision and goals.

Project never exceeded six months, and was sometimes as short as two months.

Hardware of some kind was in place from the beginning of the project. This was not always the hardware eventually used for the target or development system.

Making the delivered prototype version one, or the first increment, of a delivered system frequently failed. Case A is the exception supporting the rule.

Off-the-shelf software, inherited code or architecture, or development tools (or all of the above) were used and considered an essential element in successful delivery.

Documentation was minimal, but some was considered essential. Documents most often cited were:

(1) Initial project plan, telling what is going to be tried, and how (e.g., the "Prototype Development Plan" of Figure 4). This may be very brief, and included in the task plan.

(2) Design document, containing functional model and architecture that worked, plus those that did not work, and why (briefly).

(3) Proposed user requirements document, with requirements semi-ordered by priority (several items may have the same rank).

The only review was usually a final demonstration.

3.2.2 Evolutionary Type

Most often used incremental development, with design and requirements being modified, based on experience with increments already delivered. This was still a learning process.

The extent to which prototyping was used, if at all, depended on other environmental factors.

Required intelligent user involvement.

Maximum development lifetime before system succumbed to lack of flexibility and robustness in the evolving design was about two years. Designs were often limited by initial choices based on early requirements.

Times for developing one increment ranged from six months to one year; less than six months was not considered sufficient for a robust implementation of a moderately complex function.

Current documentation is especially important in this type of development environment; out-of-date documents causing confusion and misunderstanding. Documents most often cited were:

(1) User requirements, with those not yet implemented ordered by priority.

(2) High level design, including architectural design, and overall system model or concept (the goal the system is evolving toward).

(3) Detailed design, including interface specification.

(4) User guides, sometimes used as supplemental design documents for increments or functional elements.

(5) Integration and test plan.

(6) Data dictionary where appropriate to the application (including terms, data names, and variable names).

Other documents may be required for specialized systems (e.g., large database systems).

Reviews were usually held for requirements, design, and delivery of each increment. Code walkthroughs were used extensively during implementation, for coordination as well as detecting problems.

3.2.3 Established Type

This is the type covered by conventional application of the JPL Standard [7].

Minimum development time to produce a robust system for use by other than the developer's organization seemed to be about two years.

This type required the least interaction with the sponsor and user, because a mature, well specified set of requirements existed.

Hardware procurement often proceeded concurrently with requirements definition and high level design.

Documentation fulfilled its primary function of supporting system upgrade and maintenance.

Integration test and acceptance test, and their associated plans, were significant parts of the development cycle.

3.3 PRELIMINARY CONCLUSIONS

(1) Highly tailored standards are needed for experimental and evolutionary project types, including criteria for determining project type. Documentation requirements should take into account that projects evolve from one type to the next (experimental to evolutionary to established). The documentation for each type should be designed to smoothly integrate into that for the "next" type.

(2) If a project has been planned to progress from one development type to the next, it should do so within planned time limits, and these time limits should reasonably agree with the normal lifetime of the pertinent development type. Projects which do not progress from one development type to the next as planned should be considered for termination.

(3) Configuration management provides the feedback control necessary for a system to emerge from evolving requirements. It controls the endless, perhaps even oscillating, change possible when each delivered increment sparks a "now if we could only have" gleam in the sponsor's eye, and each requested change gives the profes-

sional software developer an opportunity to make the system "even better" technically.

(4) Support needs to be provided to projects in analyzing their project environment and choosing the software development paradigm best suited to it.

(5) Training is needed to help individuals work within and feel comfortable with the development paradigm chosen for a project.

(6) Projects for which off-the-shelf software and development tools were needed most, often to meet tight schedules, did not feel they could risk allotting time to search for, evaluate, or acquire these tools. Support in assessing the needs of the project, selecting the best tools, and acquiring those tools, is needed. This support could be provided either centrally, or through a network of organizations and consultants. Varying levels of support, in terms of time and thoroughness of analysis, should be provided, because different projects have very different needs and resources. The steps to take to obtain the level of support a project needs should be well publicized.

(7) For prototyping and incremental approaches to be successful as methods to develop and refine requirements, close cooperation between sponsor and developer is required. This creates an environment in which sponsor and development personnel can learn more about the uses of automation in the application domain. If it appears that this cooperative environment can not be established, serious consideration should be given to whether the project should be undertaken.

(8) Both development and integration and test hardware are needed to take advantage of the opportunities to shorten development schedules provided by an incremental approach. This allows development to start on the next increment while integration and testing is being completed on the current increment.

REFERENCES

1. **Distributed Management Information and Control System (DMICS): Lessons Learned**; JPL Internal Document D-4940, October 28, 1987.

2. **Incremental Development and Prototyping in Current Laboratory Software Development Projects: Research Plan**, Griesel, M. A.; (JPL Internal Project Report) February, 1988.

3. **Joint Exercise Support System (JESS) Executive Overview**; JPL Internal Document D-3917, January, 1987.

4. **Joint Exercise Support System (JESS) Software Management Plan**; (JPL Internal Project Report) February, 1987.

5. **Joint Exercise Support System Technical Summary**; (JPL Internal Project Report) February, 1988.

6. JPL Highlights 1987 - The Global Decision Support System; Memorandum 3630-87-080, de Gyurky to Bane, dtd. August 11, 1987.

7. **JPL Software Management Standard, Version 2.0**; JPL Internal Document D-4000, December, 1987.

8. **New Paradigms for Software Development**, Agresti, W. W.; IEEE Computer Society Press, Washington, D. C., 1986.

9. **Planetary Data System Project Plan**, Renfrow, J. T.; JPL Internal Document D-3492, May 2, 1986.

10. **Planetary Data System Software Management Plan, Revision 1.0**; JPL Internal Document D-3487, August 10, 1987.

11. "Prototype Proposed Charter"; from **SFOC Prototype Interim Report No. 8**, Appendix A, JPL Internal Document D-4574, July 15, 1987.

12. **Space Flight Operations Center Prototype Phase 1 Summary Report**; JPL Internal Document D-3340, May 16, 1986.

13. **Research Methods in Social Relations,** Sellitz, C., Jahoda, M., Deutsch, M. and Cook, S. W.; Holt, Rinehart and Winston, New York, 1959.

14. "An Alternative Prototyping Classification," Mayhew, P. J., and Dearnley, P. A.; **The Computer Journal** 30 (1987) 481-484.

15. **Software Engineering: A Practitioner's Approach**, Pressman, R. S.; McGraw Hill, New York, 1987.

16. "A Spiral Model of Software Development and Enhancement," Boehm, B. W., **IEEE Computer**, May, 1988, 61-72.