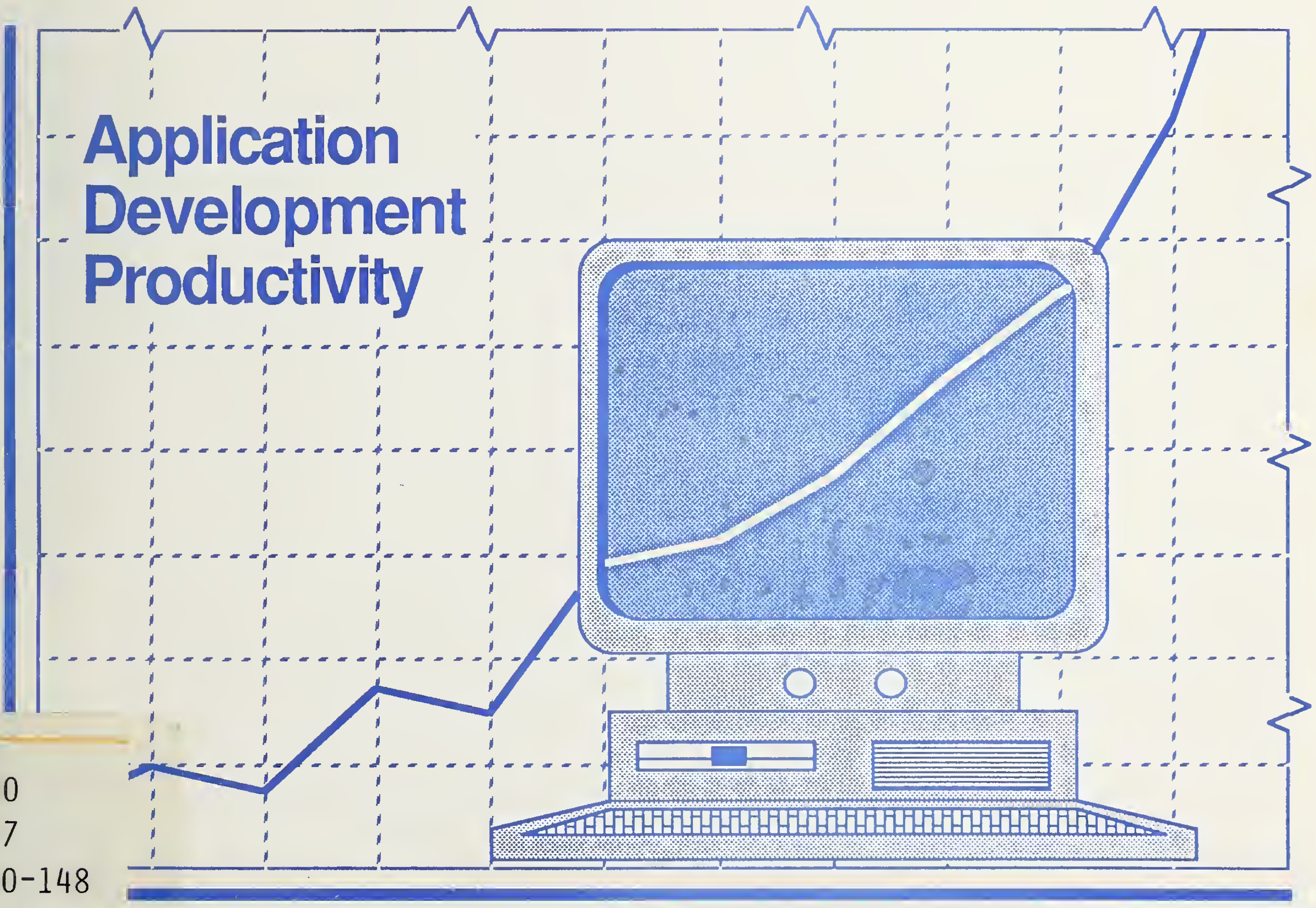
Computer Science and Technology

NBS Special Publication 500-148

Application Software Prototyping and Fourth Generation Languages

Gary E. Fisher



The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research to assure international competitiveness and leadership of U.S. industry, science and technology. NBS work involves development and transfer of measurements, standards and related science and technology, in support of continually improving U.S. productivity, product quality and reliability, innovation and underlying science and engineering. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, the Institute for Computer Sciences and Technology, and the Institute for Materials Science and Engineering.

The National Measurement Laboratory

Provides the national system of physical and chemical measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; provides advisory and research services to other Government agencies; conducts physical and chemical research; develops, produces, and distributes Standard Reference Materials; provides calibration services; and manages the National Standard Reference Data System. The Laboratory consists of the following centers:

- Basic Standards[2]
- Radiation Research
- Chemical Physics
- Analytical Chemistry

The National Engineering Laboratory

Provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

- Applied Mathematics
- Electronics and Electrical Engineering[2]
- Manufacturing Engineering
- Building Technology
- Fire Research
- Chemical Engineering[3]

The Institute for Computer Sciences and Technology

Conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following divisions:

- Information Systems Engineering
- Systems and Software Technology
- Computer Security
- Systems and Network Architecture
- Advanced Computer Systems

The Institute for Materials Science and Engineering

Conducts research and provides measurements, data, standards, reference materials, quantitative understanding and other technical information fundamental to the processing, structure, properties and performance of materials; addresses the scientific basis for new advanced materials technologies; plans research around cross-cutting scientific themes such as nondestructive evaluation and phase diagram development; oversees Bureau-wide technical programs in nuclear reactor radiation research and nondestructive evaluation; and broadly disseminates generic technical information resulting from its programs. The Institute consists of the following Divisions:

- Ceramics
- Fracture and Deformation[3]
- Polymers
- Metallurgy
- Reactor Radiation

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Gaithersburg, MD 20899.
[2]Some divisions within the center are located at Boulder, CO 80303.
[3]Located at Boulder, CO, with some elements at Gaithersburg, MD

Computer Science and Technology

NBS Special Publication 500-148

Application Software Prototyping and Fourth Generation Languages

Gary E. Fisher

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Gaithersburg, Maryland 20899

May 1987

U.S. DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Bureau of Standards
Ernest Ambler, Director

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-148
Natl. Bur. Stand. (U.S.), Spec. Publ. 500-148, 65 pages (May 1987)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 87-619824

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1987

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington DC 20402

TABLE OF CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

Application prototyping is a technique used to define requirements and specify constraints on a proposed software system. It is based on the premise that, in a wide range of problem domains (particularly on-line interactive systems) users of the proposed application do not have a concrete idea of what the application should do, nor how it should operate. This report is designed as an introduction to the planning, organizing, executing, and controlling of a methodology for application prototyping. Senior level technical personnel and project management who are interested in instituting a prototyping program will benefit from this information.

The methodology combines application prototyping and Fourth Generation Languages (4GLs) into a cogent and viable software life cycle designed to acknowledge the inherent difficulty in specifying software requirements. Often, after a system is operational, errors or shortcomings overlooked during development become evident. Application prototyping attempts to overcome these problems by providing users and developers an effective means to communicate ideas and requirements before a significant amount of effort has been expended in development. The prototyping process results in a functional set of specifications that can be fully analyzed and understood by users, developers, and management in deciding whether the application can be developed, and how it should be developed.

The influx of Fourth Generation Languages has induced many organizations to undertake projects based on prototyping techniques. 4GLs provide many of the capabilities necessary for prototype development, such as those described in NBS Special Publication 500-138, **A Functional Model for Fourth Generation** Languages. These capabilities include user functions for defining and managing the user/machine interface, data management functions for organizing and controlling access to data, and system functions for defining execution control and interfaces between the application and its physical environment.

The benefits of the proposed methodology are the abilities to 1) identify requirements and problem areas in early development phases, 2) reduce the development time through reuse of the prototype or through knowledge gained in developing and using the prototype, and 3) produce systems based on true, rather than perceived, requirements.

The combination of application prototyping and Fourth Generation Languages provides a cost-effective and controllable method of developing and maintaining software.

1. INTRODUCTION

Software prototyping is emerging as a viable software engineering technique. A recent exchange over an electronic mail system highlights the issues of prototyping as follows:

> *[Initial message] "All too often, one sees programmers writing code before a proper job of analysis and design has been done. I also believe that is partly because semi-running code makes it appear as though progress has been made, while a complete design doesn't convey the same impression."*
>
> *[Response] "All too often, one sees programmers writing detailed design specifications before writing any code. This is probably because design specs make it appear that the problem is fully understood, and give the impression to management that the rest of the process of implementation will be entirely mechanical and hence will be on budget and on schedule. ... Then one gets to draw up a new budget and schedule for 'maintenance", which is the process of modifying the program so that it really meets the customer's needs, instead of merely meeting the specifications.*
>
> *The alternative is to recognize that (a) the user probably does not have a complete and coherent idea of what he needs, and hence cannot write a spec or meaningfully assess one you write, and (b) in any case, the presence of the software itself will change the user's tasks and therefore his needs. Given recognition of this situation, it is not even theoretically possible to avoid a trial-and-error process of software development. Hence you should aim to make your inevitable mistakes as early as possible. Which puts a heavy premium on getting initial prototype software into the hands of the customers right away, so that you can learn what's wrong with it. One progresses by iteratively enhancing (and perhaps sometimes re-doing) the prototype, with regular user feedback.*
>
> *This is not to say that the design-it-first method doesn't have its uses, and its advantages, when the problem is understood well enough. But a very large class of problems--almost anything to do with user interaction, for example--simply don't meet that criterion."* [Spen86]

Development of automated systems based on techniques of prototyping is not a well-understood subject. This report is an attempt to shed some light on the process of prototyping by describing what prototyping is, pros and cons of the concept, how it affects the life cycle of software, and the implementation of a life cycle that integrates prototyping. While the report is designed primarily for use by senior technical and mid-management personnel in developing an organizational approach to prototyping, it is hoped that software developers can garner ideas to reduce the effort involved in software development. In this manner, perhaps productivity and quality in developing automated systems can be enhanced.

"Software development productivity" is one of the most fashionable topics in the information industry today. Large amounts of resources have been and will be expended by government and industry in an attempt to overcome the problems associated with productivity and quality in the development and maintenance of software.

James Martin stated in his book, *Application Development Without Programmers*,

> *"If we assume no increase in programming productivity ... in 10 years' time the industry will need 93.1 times as many programmers as now. There are approximately 300,000 programmers in the United States today. That suggests about 28 million programmers in 10 years' time..."* [Mart82]

The same argument was used long ago by nay-sayers against the introduction of the telephone into the majority of homes in the U.S. (i.e., that everyone in the U.S. would have to become a telephone operator). Martin also stated that the number of applications will increase 41 times over the next 10 years. These figures suggest that productivity will have to be increased by two orders of magnitude just to keep up!

Martin's predictions may or may not come true. However, the underlying assumption of this excerpt is the need for orders of magnitude increases in the production of high quality software of all types.

Some inroads to productivity have been paved, but the resulting gains have fallen far short of expectations and needs. In the late seventies, "structured programming" became a subject of intense interest in the programming world. The application of this concept resulted in a 15 to 25 percent productivity increase [Jone77, Jone79]. However, it did not attack the problem of productivity in the software life cycle as a whole. It concentrated on detailed design and coding of programs which amounted to only 20 to 30 percent of the entire life cycle effort. The other phases were not affected, except perhaps the maintenance phase. (Structured programs tended to be more readable and organized, therefore maintenance was conceivably somewhat easier.)

McCracken and Jackson wrote a short paper published in the ACM SIGSOFT Software Engineering Notes that delved into what they suspected was the root of the productivity problem. They argued that life cycles did nothing to enhance communication between users and developers; did not recognize end users as potential developers; did not involve users past the definition phases; and did not recognize that requirements have never been susceptible to prespecification [McCr82].

Bernard Boar wrote in his book, *Application Prototyping: A Requirements Definition Strategy for the 80's* [Boar84], that programmer productivity as measured in lines of code has increased by a factor of 15 to 1 over the last 30 years. This is low compared to the 1,000,000 to 1 gain in hardware as measured in the number of transistors per chip or instruction executions per second cited for the same period. However, a major point made by Boar was that software productivity was not hindered by the slowness of creating code, but the necessity imposed by life cycles to prespecify requirements.

Prespecification implies that all system specifications are known and can be rigorously defined before the system is implemented. While this may be desirable, it is often not possible. Prespecification stultifies users and developers alike: users, because they do not have concrete ideas of what they want and are continually changing the requirements; and developers, because they cannot cope with what are simple changes in the eyes of the end users.

The act of implementing a system is a learning experience for the system user and usually leads to new information about the system that was not known prior to implementation. The result of the new information leads to the need to specify changes, perform impact analyses due to the changes, and cost-justify resources to make these changes. This is particularly discouraging to the end user when the identified changes have profound effects upon the usability of a system, but cannot be economically justified from management's perspective. One suggested means of overcoming the problems associated with life cycles and prespecification is to develop specifications and systems based on prototypes.

A prototype of an automated information system is an experimental version of the system. It may be incomplete by not providing all of the functionality required. It may be modified many times in direct response to user feedback. It must be inexpensive to build in comparison with development of the full system. It must be easily modified, and it must be flexible to the point that it could conceivably be expanded to fulfill all system requirements. Because it is initially experimental or explorative, it might be thrown away. Prototypes have been used for centuries in the engineering of buildings, bridges, airplanes, and other mechanical systems to verify assumptions made about the full-scale project. Many times, prototypes display physical flaws that cannot be seen in the written specifications. Through this process of model building, the users and implementors can come to agreement on what the final product should be.

Prototyping, however, is antithetical to the classical software development model. Prespecification and rigorous control of changes form the basis of most life cycles. Prototyping works in an alternate world where change and experimentation are desirable. The problem is how to incorporate prototyping into a manageable development model without disrupting the effectiveness of prototyping or the control needed to manage development. This calls for a rethinking of the way in which software is developed.

One of the promising tools in support of prototyping is Fourth Generation Languages (4GLs). A 4GL is a software system of integrated tools designed to assist end users in developing interactive online business applications with a minimum need for knowledge of the technical aspects of data processing. Components of a typical 4GL are a database management system (DBMS), a query language, a report generator, a screen formatter, a data dictionary, and a high level procedural language. Using a 4GL allows end users and developers to put a prototype together in days or weeks as opposed to weeks or months with traditional programming languages, make changes to the prototype as they are required, and emerge with a reasonable facsimile of the finished product [Fish86].

The remainder of this paper discusses the pros and cons of prototyping, provides a short guide to current thoughts on prototyping, and presents a methodology for utilizing prototypes in software development. Section two describes the features of prototyping and contrasts it to prespecification. Section three presents a guide to several prototyping models and a proposed model based on specification prototyping using a Fourth Generation Language. Section four summarizes this report and presents conclusions on the prototyping based model and the use of 4GLs as a prototyping vehicle.

2. SOFTWARE PROTOTYPING

"Prototyping is ...
flying by the seat of your pants ...
iteratively!" [Aqui85]

There are various types of software prototypes. They range from paper descriptions of inputs, processes, and outputs, to fullblown automated versions. They may take anywhere from several hours to several years to build, and may cost several hundred dollars to several million dollars.

An exact definition of software prototype is impossible to find. The concept is made up of various components. A prototype--

- is an actual working system for experimenting and from which lessons can be learned in revising requirements [Goma83];
- must be comparatively cheap to build (less than 10 percent of the full system development cost) [Goma83, Bric83];
- must be relatively quickly developed so it can be evaluated early in the life cycle [Goma83];
- does not eliminate or reduce the need for comprehensive analysis and specification of user requirements [Goma83];
- provides users with a physical representation of key parts of the system before implementation [Boar84, Bric83];
- is not necessarily representative of a complete system [Bric83];
- performs only a subset of the functions of the final product [Zave82]; and
- lacks the speed, geographical placement, or other physical characteristics of the final system [Zave82].

William Riddle expressed the term "prototyping" most succinctly as follows:

> *"[Prototyping is] the building of trial versions of software systems that emphasize the preparation of immature versions that can be used as the basis for assessment of ideas and decisions in preparation of a version that is complete and deliverable."* [Ridd83]

2.1. Benefits of Prototyping

In the last few years, the recognized benefits of prototyping have become relatively numerous. A few of these benefits cited by various authors are listed as follows:

- o Prototyping emphasizes active physical models [Boar84] (i.e., The prototype looks, feels, and acts like the real system.)

- o Prototyping is highly visible and accountable; therefore, management of prototyping is simple compared to full-scale implementation [Boar84].

- o Several burdens are eliminated in prototypes: performance, optimum access strategies, and complete functionality [Boar84]. As a result, the prototype should not require as much effort in development as compared to that of the full system.

- o Issues of data, functionality, and user/machine interfaces can be readily addressed [Boar84].

- o Requirements can be separated from system specification and decisions can be made about what to prototype and what not to prototype [MacE82].

- o Users are usually satisfied, because they get what they see [EDP84].

- o Many design considerations are highlighted and a high degree of design flexibility becomes apparent [Bric81].

- o Information requirements are easily validated [Appl83].

- o Changes and error corrections can be anticipated and made on the spur of the moment in many cases [Appl83].

- o Ambiguities and inconsistencies in requirements become visible and correctable [Goma83].

- o Useless functions and requirements can be eliminated [Goma83].

2.2. Disadvantages and Limitations of Prototyping

While the advantages of prototyping are many and valuable, the disadvantages and limitations of prototyping deserve consideration. Some of these are--

- o Because the prototype is not the complete system, analysis of requirements may not be adequate. Symptoms of the problems may be mistaken for the problems [EDP84]. (e.g., A user may demand online access to daily data when the solution is actually a timely summary report.)

- o A prototype system may evolve into a production system before it is ready to be used [EDP84].

- o Prototyping is still viewed by management as development. Throw-away (i.e., prototype) code is not considered to be economically justifiable [Boar84]. (There should be no stigma attached to throwing away the prototype if the possibility for such action is planned.)

- Because it is heuristic in nature, prototyping is not as reliable as more rigorous approaches to requirements specification and analysis [Boar84].

- The technology of prototyping is less well-understood and less available than other means of system development (i.e., The tools and techniques are not as wide-spread as those of other software development techniques.)

- End users make unrealistic demands based on prototypes [Smit85].

- Prototypes do not address security procedures, backup/recovery, conversion, system testing, implementation plans, user sites, system reliability, controls and documentation, and training [Adam85].

- The actual performance and ease of maintenance of the finished product cannot be ascertained from the prototype [Tayl82].

Some of these limitations and disadvantages can be refuted as follows:

- Analysis of requirements may not be adequate in prototyping. However, in general a prototype is not the final system, and as stated before, is not a replacement for full analysis. Prototyping experts do agree that users are happier with the delivered system when they have a great deal of contact with the development staff during evolution of the prototype. Contrast this aspect with systems that have failed, even when based on rigorous techniques of system prespecification.

- The purpose of a prototype is to assist in defining requirements and analyzing specifications. As with all technical fields, prototyping requires understanding and training before implementation should proceed. There are documented cases where users did not want to give up the prototype system [EDP84], even though the prototype may have been inefficient or an incomplete system. The only way to combat this possibility is to educate the users on the purpose of the prototype and provide controls to manage prototype evolution. However, if the overriding requirements of the users can be met with the prototype, and no one objects, then perhaps no problem exists in using the prototype as the delivery system.

- Proponents of rigorous approaches to software development often include methods that rely heavily on labor intensive documentation of the specifications and plans. More often than not, these methods are more heuristic in nature than is prototyping. For example, the typical waterfall model of the software life cycle is not based on any mathematically proven or correct engineering principles. Prototyping is heuristic, but it allows for experimentation and insight into the human-computer domain. Traditional approaches do not take this into consideration.

- One of the major arguments against prototyping in the past was that prototype systems were almost as expensive and just as difficult to modify as the production systems. Fourth Generation Languages (4GLs) have helped overcome this problem to a large extent. Throw-away code is economically feasible when 4GLs are used for prototyping. In the sense that a 4GL can be used to produce a prototype with relatively much less effort than is needed when using a third generation langauge (3GL) such as COBOL or C, the 4GL prototype may be thrown away. To throw away a prototype written in a 4GL is much more easily justified than for a prototype in a 3GL.

- End users always have, and always will, make unrealistic demands on systems developers. (Or is it "Systems developers always produce the wrong system?")

- Prototypes do not address security, backup, testing, controls, etc. This argument is based on environmental factors more than on the concept of prototyping. In truth, prototypes can encompass many of these aspects according to the tools available for creating the prototypes. Some 4GLs provide capabilities to directly address factors affecting security, backup/recovery, and auditing. The users' acts of iteratively modifying and reviewing a prototype is a type of testing that is not even prescribed in most life cycle models. Documentation, training, and maintenance of the system are enhanced because of direct user involvement at the outset.

- Performance of 4GLs has been debated over and over. While many 4GLs are interpretive and have an associated higher overhead in operation, many vendors are reimplementing their products in compiler versions and optimizing execution. This is a direct result of competition among 4GL vendors. Now, instead of executing ten times as slow as systems in third generation languages, they approach very close to execution times of third generation systems. Very large applications are being produced by organizations such as the Federal Bureau of Investigation, the Social Security Administration, and the Customs Service that include online query, distributed data, high volumes of transactions, and large databases all of which are handled by 4GLs.

3. PROTOTYPING METHODOLOGY

A methodology consists of the methods, rules, and procedures organized to solve problems in a specific domain. If the methodology has developed successfully, these methods, rules, and procedures should be well-integrated. The following subsections describe several life cycle integrated prototyping methodologies that have been proposed in recent years. They are presented to show the diversity of concepts involved in defining software life cycles and to illustrate the effects of prototyping on the life cycle in general. The recurring theme throughout the majority of the models is that the prototype is one small part of the development cycle, but that its ramifications are felt overall. The models are presented in alphabetical order by author's name.

3.1. Appleton

Data-Driven Prototyping consists of the following six steps. They are--

o **Operational review** - define the project scope; evaluate the environment, current organizational, and information structures;

o **Conceptual design** - define proposed metadata (i.e., the structure of data and relationships between individual structures); scenarios to describe service functions that change data states; and types of retrievals;

o **Data design** - normalize the metadata;

o **Heuristic analysis** - check consistency of requirements against metadata through the use of real data values and iterate this with the data design step;

o **Environment test** - build programs to support data entry and retrieval (prototype); and

o **Monitor performance** and tune the application [Appl83].

3.2. Boar

Bernard Boar's book, *Application Prototyping: A Requirements Definition Strategy for the 80's*, provides an in-depth discussion of many aspects of application prototyping. He proposes the following model:

o **Identify basic needs** - This phase concentrates on identification of fundamental goals and objectives, major business problems to be solved, definition of data elements, data relations, and functions.

o **Develop working model** - Quickly build a working prototype that addresses the key needs identified in the previous step.

- **Demonstrate the prototype** - Present the prototype to all interested users and obtain additional requirements through user feedback. Stimulate the users with "what if" types of questions.

- **Prototype is done** - Iterate between demonstration and enhancement of the prototype until the users are satisfied that the organization could provide the service needed from the prototype. Once the users agree that the prototype fits the concept of the service needed, the prototype can be enhanced into the final system, or rewritten in a less resource-consuming language [Boar84].

3.3. Connell and Brice

This model replaces the traditional life cycle phases with a **rapid prototyping** process. The steps are--

- **Rapid analysis** - results in an incomplete paper model that shows the system context, critical functions, an entity-relationship model of the database, and conceptual tables, screens, attributes, reports, menus, etc.;

- **Database development** - uses a relational architecture to create a working database for the use of the prototype;

- **Menu development** - expands upon the initial concepts defined in rapid analysis and fixes the hierarchical structure of the application;

- **Function development** - functions are grouped by type into modules;

- **Prototype demonstration** - iterate by redoing parts as necessary and tuning where possible;

- **Design, code, and test** - the detailed "as built" design specifications are completed; and

- **Implementation** - based on the evolution of the prototype and completion of all programs, tests, documentation, etc. [Conn84]

3.4. Dearnley and Mayhew

P. A. Dearnley and P. J. Mayhew presented their ideas on software tools in prototyping at a working conference on prototyping in 1983 [Dear84]. Essentially their concept on software development with prototyping consists of two separate but interrelated cyclic models: one consisting of a classical software development cycle and the other a prototyping cycle that interacts with the classical model during the phases of analysis and design. The diagram of these two models resembles a figure eight lying on its side. (See figure 1.) The major operations and operators are:

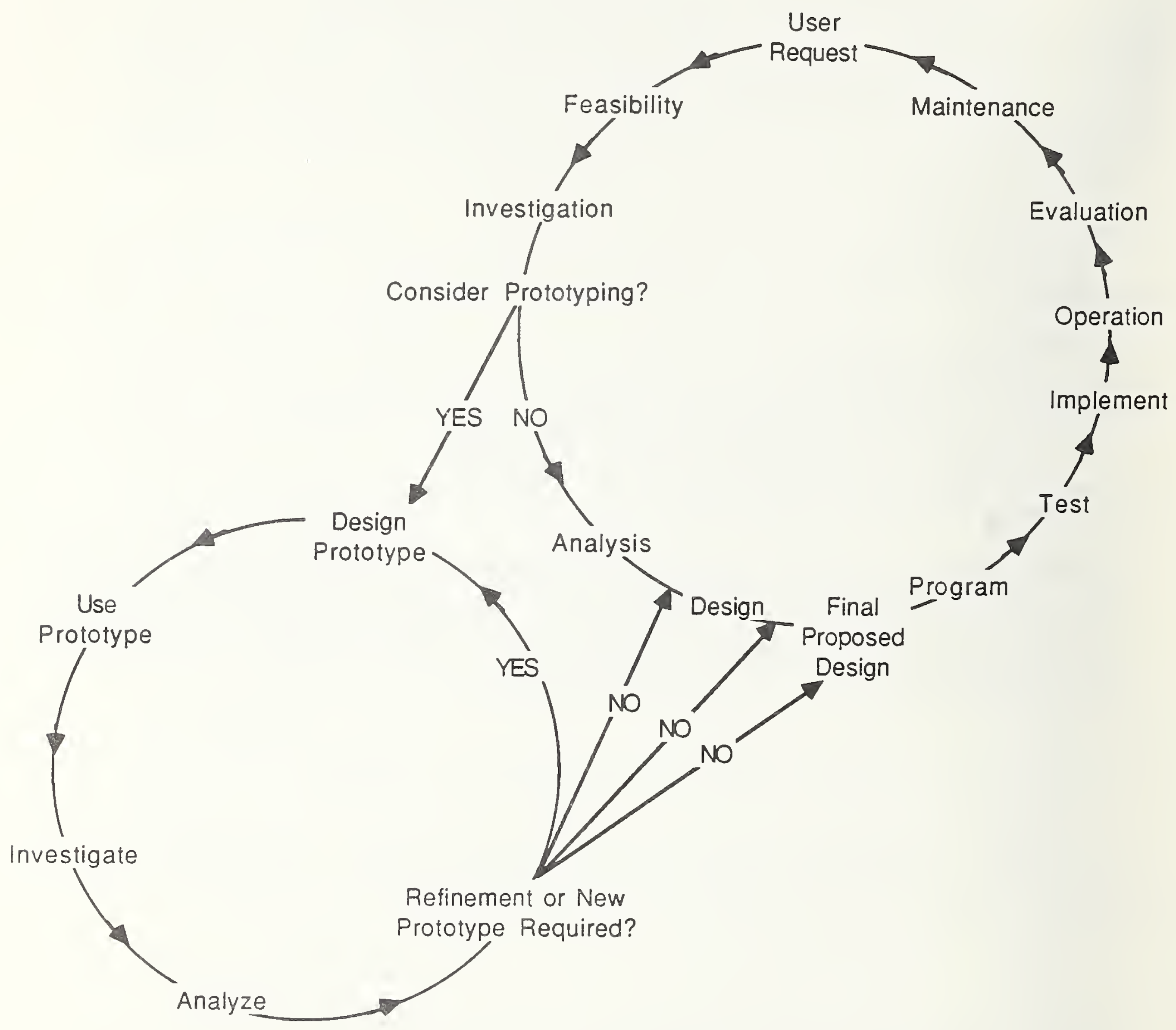
- o **Classical** cycle
  - User request
  - Feasibility
  - Investigation
  - Consider prototyping (see prototyping cycle below)
  - Analysis
  - Design
  - Final proposed design
  - Program
  - Test
  - Implement
  - Operation
  - Evaluation
  - Maintenance

  (Repeat the cycle)

- o **Prototyping** cycle
  - Design prototype
  - Use prototype
  - Investigate (Use prototype some more)
  - Analyze (Investigate prototype some more)
  - Refinement, or is new prototype required?

  (Repeat the cycle if not done)

The interaction of the two cycles occurs when investigation in the classical cycle uncovers the need to prototype, at which time the prototyping cycle is entered. Prototyping is terminated when analysis, design, or the final proposed design of the classical cycle can be completed based on information discovered or verified in the prototyping cycle.



Concept used by permission of the authors and publishers.

Figure 1. Dearnley and Mayhew Prototyping Model

3.5. EDP Analyzer

The EDP Analyzer's Special Report - Fourth Generation Languages and Prototyping, proposes the following life cycle:

- o Create a prototyping team of one analyst/programmer and one end user.
- o Identify the user's basic needs by interviewing several end users to define the problem and sample user expectations.
- o Quickly develop a prototype that addresses most of the issues of the problem and user expectations.
- o Demonstrate the prototype to the end user. If it fits, let the user experiment with it and perform work. Otherwise, scrap it and try another prototype. Specify a time period in which the user may use the prototype.
- o Refine the prototype by including changes identified through use. Iterate through this step and the previous step until the system fully achieves the requirements.
- o Identify an end user test group to get more feedback on the prototype. Allow this test group a specified period of time to exercise the prototype.
- o Determine whether the prototype will be implemented, or the system will be rewritten in a conventional language. Base this decision on maintenance considerations, hardware/software efficiency, flexibility, and other system requirements [EDP84].

3.6. Gomaa

This model is proposed to assist in specifying user requirements, verifying the feasibility of system design, and in translating the prototype into the final system. The procedure definition follows:

- o Preliminary analysis and specification of requirements to establish a baseline for future reference.
- o Design and implementation of a prototype emphasizing the user interface, small development team, prototype development language, and tools to assist in rapid development.
- o Exercise the prototype in the user's workplace.
- o Refine the prototype by incorporating user comments as quickly as possible.
- o Refine baseline requirements by incorporating lessons learned from the prototype.

- Design and implement the production system using a traditional life cycle with requirements derived from the prototype [Goma83].

3.7. MacEwen

The **Iterative Development Accounting** life cycle is based on the view that a system is a sequence of specification levels with an increasing amount of detail at each level. These levels are--

- informal requirements;
- formalized requirements;
- design;
- implementation;
- configuration; and
- operation.

Each level contains more detail than the one above it. Additionally, each level must be balanced with upper level specifications. Iterative Development Accounting imposes development accounting on each level. This means that a change in a discrete level of specification can only be made if the next higher level has been modified to accommodate the change. In turn, the previous level can be changed only if the level above it has been modified to allow the lower level modification.

A complete history of development is maintained by this accounting technique to insure that consistency remains throughout all levels. A prototype is developed at each level to show that the specifications are consistent. Each prototype concentrates on the functions to be evaluated at that level. The final prototype becomes the implemented system once testing, installation, and training have been completed [MacE82].

3.8. McCracken and Jackson

Two models are presented here. The first is termed the **evolutionary model** in which the prototype is built and gradually enhanced to form the implemented system. The second has become known as the **"throw one away"** model (after Fred Brooks' expression in his book, *The Mythical Man Month* [Broo75]).

The end user becomes an integral part of the prototype development in both models. The authors suggest that the end users be trained in the use of a prototyping tool, such as a simulation language or a Fourth Generation Language. The two models are described briefly as follows:

o Method 1

  - In responding to the end user's earliest and most tentative needs, allow the user to experiment with and use the prototype to perform productive work.
  - The analyst watches the user to see where fitting of the prototype needs to take place. A series of prototypes, or modifications to the initial prototype, could then evolve into the final product.

o Method 2

  - Implement a prototype. Write the initial design from this and the end user's feedback. Produce another prototype to implement the initial design. Design the final system and implement in a conventional language [McCr82].

3.9. Wasserman

The User Software Engineering Methodology (USE) is based on a model of software development that is partially formal and partially informal. Parts of the methodology have been automated in the Unified Support Environment. The USE methodology includes the following steps:

o Requirements analysis includes activity and data modeling, and identification of user characteristics.

o An external design step develops transactions, and user/program interfaces.

o A "facade" of the system is developed as a prototype of the user/program interface and is revised as needed.

o Narrative text is then used to informally specify the system operations.

o A preliminary relational database is designed as the basis for a functional prototype of the system.

o The functional prototype is developed to provide at least some of the functionality of the proposed system, and perhaps all.

o A formal specification of the system operations may be optionally developed at this point.

o The system architecture and modules are designed.

o The system is implemented in PLAIN (a procedural language used in the Unified Support Environment).

o Testing and verification are performed on the implemented system before being released into the production environment [Wass83].

3.10. Young

This prototype methodology describes the development of a system as the evolution of a prototype. The stages are described as follows:

- Stage 1 - Management states the organization's objectives in terms of information requirements, and the scope of the system boundaries and capabilities. Prototype screens and reports are developed.

- Stage 2 - The end users and management review and approve the prototype. Full system design, equipment selection, programming, and documentation are completed.

- Stage 3 - Management reviews and commits to implement the system. System tests of the prototype are run in parallel to the old system. Work begins on the next release which causes an iteration of all three stages [Youn84].

4. PROTOTYPING STRATEGIES AND FACTORS

The models discussed in section three suggest numerous strategies for prototyping. The life cycle can be modified to add a prototyping phase, or prototypes can be built during several phases. The prototype may be thrown out when no longer needed, or it may evolve into the finished product. In each organization that chooses to implement a prototyping life cycle, various factors must be taken into account in deciding what strategy to follow. Some of these factors are--

- o personnel resources;
- o applicability of prototyping;
- o hardware constraints; and
- o availability of prototyping tools.

These factors are examined in the following subsections to define the context for prototyping in a Fourth Generation Language.

4.1. Personnel Resources

One of the most important aspects of implementing prototyping is to ensure that the technical personnel who will develop prototypes are fully trained in the techniques and control of prototype development. They must also be able to teach end users what they need to know about prototyping.

Technical prototype developers must be thoroughly knowledgeable in the methodologies applied to current software life cycles. They must be experienced particularly in software development, requirements specification, and analysis. Junior programmers must be guided by more experienced and senior personnel who are very familiar with the application of prototyping and the constraints on a particular development effort.

Additional characteristics include the ability to make presentations before senior management, such as in demonstrating prototypes for the benefit of decision-makers, and experience in estimating costs and project schedules.

4.2. Applicability of Prototyping

Inevitably the question, "What should be prototyped?", must be answered. The answer is dependent on several factors such as what types of applications will be developed, and the eventual use of the prototype in relation to a particular application. These and other areas are addressed in the following subsections.

4.2.1. Types of Prototypes

Christiane Floyd described several classes of prototypes in a paper entitled "A Systematic Look at Prototyping" [Floy83]. These are--

- Explorative prototyping which emphasizes the clarification of requirements and features to be developed, and helps in deciding among alternative solutions. (This type of prototype is generally thrown away since the direction it must follow is not necessarily known in advance.)

- Experimental prototyping which assists in determining if the proposed solution is adequate before a large investment in development is made. In this case, the prototype consists of any number of proposed functions, the user interface, or a test of the proposed system architecture. Experimental prototypes may be thrown out, or they may evolve into deliverable systems.

- Evolutionary prototyping which is performed with the express purpose of evolving into the final system. They are generally marked by releases of incremental versions that add progressively more functions and capabilities as the need arises and resources permit.

Another view of prototyping accentuates the logical and physical aspects of a system. These prototypes are known as **mockups**, **functional prototypes**, and simulations. Mockups address the user interface as the primary target. The goal of a mockup is to incorporate user characteristics and preferences in the way a system is perceived. Ease-of-use is a primary concern in this type of prototype.

Functional prototypes concentrate on specific capabilities that are required in the delivery system. The ability to program these functions and the interactions among various functions are tested and reconfigured until the appropriate mix is achieved.

Simulations are used to estimate metrics such as response times, database access times, throughput rates, and other performance characteristics required in the delivery system. Simulations are generally carried out after mockups and functional prototypes have been developed. The requirements developed in the process of prototyping the user interface and functional components of the system have a direct impact on the factors measured with simulation.

In practice, a mixture of mockups and functional prototypes should produce a realistic set of requirements. It would not be unusual to find that all three types of prototyping are used in many cases. (Section 5.3 provides more specific guidance on what to prototype after a logical data model has been completed.)

4.2.2. Application Domain

Prototyping has been used in various application systems. For example, a realtime process control system in a manufacturing context is documented in

[Goma83]. The Santa Fe Railroad developed large systems for tracking freight and 20,000 pieces of rolling stock, and for billing customers. The Federal Bureau of Investigation has used prototyping in the development of administrative personnel systems for its 9,000 agents throughout the United States. The Criminal Justice database system has been implemented partially through the use of prototyping.

There are numerous cases of large and small systems documented in the references cited at the end of this report. The range of application domains in these references appears to run the gamut of all types. Experiences have not provided conclusive evidence to say that prototyping is better suited to one or another type of application domain. However, advice reported in many references suggests that prototypes should be used for those areas of an application that are not well understood and for user interfaces.

4.3. Hardware Constraints

The primary function of a prototype is to **display functionality** of the proposed system **while searching for flaws** in the system concept. In the case where a prototype evolves into the production system, care must be taken to insure that tuning between the hardware and software is performed and monitored. This tuning, however, will occur after the decision is made to keep or throw out the prototype based on whether the prototype is explorative, experimental, or evolutionary. CPU cycles, memory, disk storage, and other types of peripherals cost money. Users of prototypes must take into account how the hardware reacts to the load caused by the prototype and the number of users to be supported by the delivery system.

4.4. Availability of Prototyping Tools

There are several general classes of prototyping tools. They include executable specification language processors, program design languages, and Fourth Generation Languages. Many of these tools have been available through public domain and commercial sources for several years.

Executable specification languages are application languages designed explicitly for defining systems. The specifications consist of commands and declarations which can be validated and verified through automated tools using various analysis criteria. These tools ensure that the system specifications are complete, unambiguous, and executable. When the user has finished documenting the system through the textual specifications, a processor may execute these text commands in emulation of the actual system. The execution can take place at various levels within the system. For example, the system may be executed at a very high level through graphical representation of the actions occurring within the system, or individual screens and reports may be executed to determine whether the content of each system object is correct.

Program design languages, such as structured English and languages found in several proprietary products, are used to define the architectural structure of a system once the requirements have been identified. Whereas specification languages are used to define "what" a system is required to do, design languages are used to define "how" the system is to implement the requirements. As such, they are generally not suited to high level functional requirements specification. Whereas specification languages may be used to define functional aspects of a system, design languages may be able to fulfill only procedural definition of the system. Design languages and functional requirements specification languages are complementary tools.

A Fourth Generation Language (4GL) is a system of integrated tools designed to assist end users in developing applications with a minimum need for knowledge of the technical aspects of data processing. A typical implementation contains components such as a DBMS, a query language, a report generator, a screen formatter, a data dictionary, and a high level procedural language. These components may be used by analysts and end users to develop prototypes by rapidly building models of data entry screens, reports, database structures, and specialized processes.

The major difference between functional requirements specification/design languages and Fourth Generation Languages is in the user interface. Because the syntax of specification/design languages are usually very complex, they are generally usable only by highly skilled data processing professionals. Fourth Generation Languages, on the other hand, are designed to be used by both data processing professionals and end users.

There are numerous commercially available 4GLs in the price range of several hundred dollars to several hundred thousand dollars. The price is usually a function of the type of machine on which a particular 4GL will execute. The larger the machine, the more functionality a 4GL is able to include.

4.4.1. Requirements for Prototyping Tools

In searching for tools to implement prototyping, some specific capabilities should be kept in mind. Required capabilities include--

- a requirements specification language to allow analysts and users to nonprocedurally describe the goals and functions of a specific software system;
- a design language to allow designers to nonprocedurally define software system components such as subsystems, modules, programs, interfaces, and control features;
- a procedural language to facilitate the programming of problems that cannot be managed in the other language subsets;
- a testing language to structure and reduce the effort needed to perform unit, integration, system, and regression testing; and
- a project management and control subset to assist in managing and controlling the development effort.

Optional but strongly recommended capabilities include--

- o an environment language to allow site managers to define the site's physical limitations and standards (e.g., how much memory and online disk storage is available); and

- o a text and graphics manipulation language for the purpose of producing documentation and man-machine dialogues.

4.4.2. Other Tool Capabilities

Other features indirectly related to prototyping tool capabilities include--

- o a specification interpreter for rapidly changing and testing specifications during development, and the ability to compile tested code for production;

- o an integrated editor;

- o a debugger that animates execution through the procedural and data management parts of the specification;

- o extensibility to allow common problem domain structures and terms to be incorporated in the specifications; and

- o optimization of database structures during analysis.

There are numerous research projects underway in academic and commercial environments to develop specification languages and tools. Each language or tool has the above capabilities as its goal. The languages, however, are still far removed from general availability. Subsets of these capabilities do exist in various forms within the Fourth Generation Languages arena. It is there that the majority of available prototyping tool capabilities will be found.

5. THE SOFTWARE LIFE CYCLE AND 4GL PROTOTYPING

The software development model presented below is based on the incorporation of prototyping. Implementation of the model is based on capabilities provided by a 4GL. A Fourth Generation Language is used to give form and content to major parts of the system prototype. In addition to textual specifications perhaps produced by other methodologies, a 4GL provides actual working models or mockups of system objects such as--

- o data entry and query screens;
- o output reports;
- o logical data descriptions; and
- o integrated procedures for demonstrating prototype processes.

The underlying concept of the model is that a prototype system is used to form the basis of the system requirements. The implemented system may then be coded in a conventional language after design has taken place, or the prototype may evolve into the implemented system through the 4GL.

Each phase of development is described, and deliverables from each phase are defined. The software life cycle based on prototyping in a 4GL consists of the following major tasks:

- o Define purpose and scope of the system
- o Develop system conceptual model
- o Develop logical data model (with assistance of a 4GL)
- o Develop a prototype in the 4GL and demonstrate it
- o Revise and finalize specifications
- o Develop the production system (4GL or 3GL)
- o Release beta test system
- o Release the production system
- o Iterate the cycle (maintain the system)

Figure 2 diagrams the procedural relationships of the phases. Figure 3 summarizes the steps and deliverables for individual tasks within the prototyping life cycle.

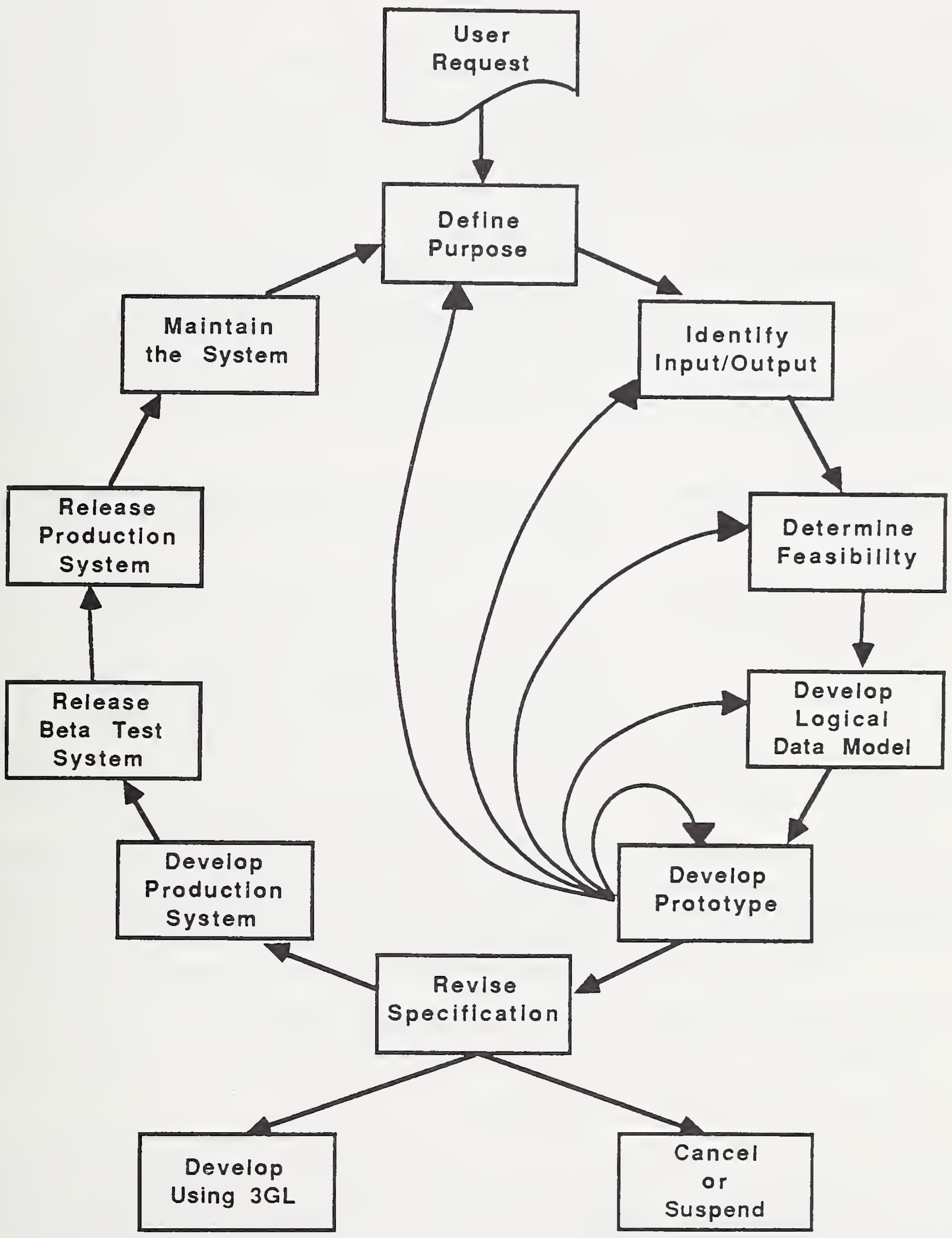


Figure 2. Software Life Cycle with Integrated Prototyping

TASK	DELIVERABLES
0. User Request	Project Request Document
1. Define Purpose and Scope of System	Statement of Goals and Objectives, Definition of System Scope, Establishment of Prototyping Team
2. Develop System Conceptual Model	System Diagrams, Data Dictionary, System Development Estimate, Prototype Estimate, Total System Life Costs, Estimate of Benefits, Risk Analysis, Screen/report Layouts
3. Develop Logical Data Model	Logical Data Model
4. Develop Prototype System and Demonstrate	Data Entry Screens, Sample Reports and Menus, Physical Database Structure, Draft User Manual
5. Revise and Finalize Specifications	Formal System Requirements Specifications, Listings of 4GL Commands, Sample Reports and Data Entry Screens, Data Dictionary Report, Actual Prototyping Costs, Revised Estimates and Schedules
6. Develop Production System	Decision: a) Rewrite in 3GL, b) Continue Evolution of Prototype, or c) Cancel the Project
7. Release Beta Test System	Revised User Manual, Error Reports, Change Requests, Detailed Design Documentation
8. Release Production System	Final User Manual, Training
9. Iterate the Cycle	(As above)

Figure 3. Software Life Cycle Tasks and Deliverables

5.1. Define Purpose and Scope of System

Why the system needs to be implemented must be defined within the context of organizational goals and information requirements. This context must include specific objectives of the system including identification of those objectives that are most critical to the success of the organization and to the system. These objectives must be balanced to preclude overly subjective or overly specific constraints that may adversely affect the system design.

5.1.1. Definition of System Scope

The scope of the system is generally based on management's objectives and the primary purpose of the system in question. The definition of system scope entails establishing the boundaries within which the system must be developed, operated, and maintained. Examples of specific factors influencing placement of boundaries include--

- o the user environment (i.e., What organizations and types of users will come into direct contact with the system?);
- o the hardware environment (i.e., What processing, storage, and communications capabilities are needed?);
- o other systems that interact with the system as sources of input and recipients of output;
- o available funding, time constraints, and resources; and
- o security of the data and software.

The purpose of a system may be further defined by requesting the proposed users of the system to describe their concept of the system: how it would be organized, and how it would operate. This can be contrasted with the results of analysis of current procedures, if they exist, to determine where changes in procedures may benefit the organization and system development.

The deliverables from this phase are a statement of management goals and objectives, and a definition of the scope of the system.

5.1.2. Establish Prototyping Team

The work of defining the major input and output, and defining a system conceptual model is performed by a prototyping team. This team consists of a senior analyst/programmer who is familiar with the business in which the organization is engaged, and one or two end users from the division that will be directly responsible for the services to be provided by the implemented system. These individuals must be trained in the techniques of prototyping and the use of a specific fourth generation language.

One of the most important factors in the success of software development based on prototyping is the understanding and enthusiasm shown by the personnel involved in the development effort. Connell and Brice [Conn85] described two prototyping projects, one a failure and the other a success, and compared various aspects of each project. They concluded that the failure was due in large part to misconceptions on the part of management and developers about prototyping and how to control it.

5.2. Develop System Conceptual Model

Using a graphical method, such as data flow diagrams (DFD) [DeMa79], or Warnier-Orr (W-O) diagrams [Orr77], construct a model of the major system functions on paper. Augment this model with data dictionary descriptions of each object contained in the diagrams. Define which objects are most critical to the success of the system by matching each object with a corresponding or closely related system objective---the more critical the objective, the more critical is the object.

The deliverables from this task are user-defined diagrams (i.e., bubble chart, W-O diagram, etc.) of the system, a text description of each object in the diagram, definitions of the major inputs and outputs of the system, and a listing of entries in the data dictionary.

5.2.1. Identify Major Input and Output

Major input and output are generally categorized as input files from other systems, manual updates, queries, output files bound for other systems, and printed reports. These are the starting points for further investigation into the user's real requirements.

The major paths of input and output through a system form a network, the nodes of which define points where transformations of data take place. These paths and the possible types of input and output form a major part of the requirements for a first-cut prototype of the system.

5.2.2. Estimate Implementation and Life Cycle Costs and Schedules

Estimate the cost of implementing the full-scale system using any method that is appropriate. Examples of cost estimating techniques are Albrecht's function points method [Albr79] and Boehm's COCOMO method [Boeh81]. The added cost of developing a prototype appears to be no more than 6 to 10 percent of the full-scale implementation cost according to current experience [Boe84a, Goma83]. For example, if the full system implementation estimate were $100,000 and 6 labor-months of effort, then the prototype cost estimate would be between $6,000 and $10,000 and would take 0.36 to 0.6 labor-months, or 1 to 2 weeks elapsed time with two persons working full time. The total development cost of the system and the prototype would then be $106,000 to $110,000.

If system development costs are approximately one-third to one-half of the total system life costs (i.e., maintenance costs are typically one-half to two-thirds of the total system life costs), then the total system costs would be in the $200,000 to $300,000 range. The prototype costs would add two and a half to five percent of the total system costs over the life of the system. However, because the prototype may allow a better grasp of the problems of design and functionality of the system, the expected maintenance costs may be less [Boe84a, Goma83, John83]. The end user exercises the prototype and can see what will be delivered. Therefore, the spate of changes that typically occur within the first six months of system use should not materialize. This is a major benefit of prototyping! [Aqui85, Boar84, Boe84a, Conn85] As a result, the total system life costs should be less using prototyping.

5.2.3. Estimate Benefits of Proposed System

Estimating system benefits is the counterpoint of system costs. Not only must development costs be recovered, but costs for maintenance support must be added to calculate a breakeven point. There are numerous methods for computing benefits from a system, but they all reduce to "How much is the functionality of the system worth?" Another way to put this is, "How much would the organization be willing to pay for the information provided by the system if it were available on the open market?"

The cost of development and support versus the value of the benefits has been neglected or disregarded altogether on numerous development projects to the combined detriment of both developers and users alike. (Most of these situations are politically motivated. Unfortunately, software development methodologies are ill-equipped to support or suppress politics.) If the costs exceed the benefits, then the system is not feasible. (Guidance on cost/benefit analysis is contained in [Chip84, Drap81, Fior83, and FIPS64].)

5.2.4. Analyze the Risk in Development

Barry Boehm expressed the main concerns in feasibility and risk analysis in an article published in the IEEE Software magazine [Boe84b]. The following are excerpts from this article:

> *"Will the specified system provide a satisfactory way for users to perform their operational functions? ... "*
>
> *"Can a system be developed that satisfies the specified requirements (at an acceptable cost in resources)?"*
>
> *"Will the specified system cost-effectively accommodate expected growth in operational requirements over its life cycle? ... "*
>
> *"Will it be cost-effective to maintain?"*
>
> *"Will it be cost-effective from a portability standpoint?"*

"Will it have sufficient accuracy, reliability, and availability to cost-effectively satisfy operational needs over its life cycle?" [Boe84b]

Have the following been considered?

"Achievable levels of overhead ... "

"Achievable levels of man-machine performance ... "

" ... Reliability of ... hardware, operating system ... "

"Availability of key personnel ... "

"Expected volume and quality of data ... "

"Expected sophistication, flexibility, and degree of cooperation of system users ... " [Boe84b]

5.3. Develop Logical Data Model

The Guide on Logical Database Design [Fong85], published by the National Bureau of Standards, defines a sequence of activities designed to produce a logical data model. These activities are described as follows:

o Model the local information flow for individual subsystems or local views of the data (i.e., for individual groups of workers or specific functional areas).

o Model the global information flow for collections of subsystems combined into a global system view (i.e., for the organization as a whole).

o Design the conceptual schema in terms of entities, relationships among these entities, and attributes that describe each entity. (This is known as the **entity-relationship-attribute** model and is independent of the structure of the system's data.)

o Model each physical external schema as a view of the conceptual schema from each individual user's perspective (i.e., in terms that individual workers would perceive).

(See [Fong85] for a complete description of the terms and methods used in logical data modeling. See Appendix A of this report for an abbreviated example of the techniques.)

After the logical data model has been produced, the next step is then to design the physical database to provide optimum access and require the least amount of storage for each user and the system as a whole.

Physical database design must always be performed before the software can access the data. The rigor of the physical database design determines efficiency. If a prototype is to evolve into a production system, it is important to access the database at the logical, not the physical, level. This allows the physical database structure to be fine-tuned as the need arises without affecting existing software. In prototyping, the developers are not worried about a precise physical model of the data, nor optimum access strategies. In many cases, the prototype is developed at the subsystem level to purposely decrease the amount of complexity involved in large system evolution.

The logical data model produced by the above steps can be integrated in large part with the prototype and used to help guide in prototype development. The modeler must approach logical data modeling from three perspectives: (1) the organization's view of the data, (2) the functions to be performed, and (3) the events that drive the system to be developed. Otherwise, valuable information about the underlying database structure could be overlooked or misinterpreted. Any data model that does not include all three perspectives is incomplete.

In developing a prototype, the prototyper can view the system in two levels of abstraction: (1) the strategic or organization-wide view, and (2) the tactical or operational view. The data modeler's perspectives and the prototyper's views must be brought together to determine what should and should not be prototyped. When the logical data model has been prepared, the developers have a base for making decisions on what to prototype.

Bernard Boar [Boar84] suggests the following guidelines in helping to determine where a prototype is appropriate:

o Systems based largely on batch processing may not be good candidates for prototyping. (These include many of the day-to-day operational systems that are functional rather than dependent on specific events or organization-wide requirements.)

o Systems oriented toward terminal operations, online database processing, and periodic or needs-based reporting would make good candidates. (This is especially true for systems that are more strategically oriented, such as management information systems; or more event driven, such as airline reservation systems.)

o Systems that take on modular structures and are functionally well-partitioned may or may not be good candidates for prototyping. An overriding concern in this case is, how much interaction will users have with the system? Usually, higher levels of interactive use point to prototyping, but this is not always true.

o The types of users involved in development of a prototype have an impact on the applicability of prototyping. If the user is uncertain about details of the system's requirements (i.e., "I don't know what I want, but I'll know it when I see it."), or the user is a decision-maker, prototyping may be appropriate. In this case, the system fits more in the categories of strategic and organizational systems. The disadvantage here is that the perfect user may not have time to participate in reviewing prototypes because he or she is too busy running the organization. If the prototype cannot be reviewed by users in a timely fashion, the project will stop dead in its tracks!

- Systems that are already in severe trouble cannot be saved by prototyping. The same goes for high pressure, "need it now" projects. Prototyping is not quick-and-dirty. It takes time and planning.

- No matter how well-intentioned are the participants, if training in the techniques of prototyping is not available, or the tools are not available, the developers should stick to prespecification.

5.4. Develop a Prototype and Demonstrate It

Using a 4GL, the prototyping team constructs a prototype system consisting of a mixture of data entry screens, printed reports, external file routines, specialized procedures, and procedure selection menus. These will all be based on the logical database structure developed in the data modeling process. The sequence of events for performing the task of developing the prototype in a 4GL is iterative. suggested procedure is described in the following:

- Define the basic database structures derived from logical data modeling. (The data structures will be populated periodically with test data as required for specific tests to be performed.)

- Define printed report formats. (These may initially consist of query commands saved in an executable procedure file on disk. The benefit of a query language in this respect is that most of the report formatting can be done automatically by the 4GL. The prototyping team needs only to define what data elements to print and what selection and ordering criteria to use for individual reports.)

- Define interactive data entry screens. Whether or not each screen is well laid out is immaterial at this point. Getting the right information in the form of prompts, labels, help messages, and validation of input is more important. (Use defaults as often as possible initially.)

- Define external file routines to process data that are to be submitted in batches to the prototype or created by the prototype for processing by other systems. This can be done in parallel with other tasks.

- Define algorithms and procedures to be implemented by the prototype and the finished system. These may include support routines solely for the use of the prototype.

- Define procedure selection menus. Concentrate on the functions performed as the user would see them rather than as the developers see them. This may entail combining seemingly disparate procedures into single functions that may be executed with one command from the user.

- Define test cases to ascertain that data entry validation is correct, that procedures and algorithms produce expected results, and that system execution is clearly defined throughout a complete cycle of system operation.

- Reiterate this process by adding report and screen formatting options, corrections for errors discovered in testing, and unambiguous instructions for the destined users. Suspend the process after the second or third iteration, or when changes become predominantly cosmetic rather than functional (e.g. when determining how many spaces should go between employee number and employee name on a data entry screen becomes a major decision).

At this point, the prototyping team should have a good feel for the overall operation of the proposed system. The team must now describe the operation and underlying structure of the prototype. This is most easily accomplished through development of a draft users manual. A printed copy of each screen, printed report, query, database structure, selection menu, and catalogued procedure or algorithm must be included. Instructions for executing each procedure should include an illustration of the actual dialogue.

5.4.1. Demonstrate Prototype to Management

The purpose of this demonstration is to give management the option of making strategic decisions about the application based on prototype appearance and objectives. The demonstration consists primarily of a short narrative description of each component of the prototype, particularly important effects of each component, and a walkthrough of typical usage of each component. Every person in attendance at the demonstration should receive a copy of the draft users manual.

The emphasis is placed on results of the prototype and its effects on tasks left to be done. At this stage, the prototype is not necessarily a functioning system, so management must be made aware of its limitations.

5.4.2. Demonstrate Prototype to Users

There are arguments for and against letting the prospective users actually use the prototype system. The major arguments against users exercising the prototype are that users' expectations are raised to an unrealistic level about delivery of the production system, and that the prototype will be placed in production before it is ready. Several cases have been documented in the computing literature of users actually refusing to give up the prototype system when the production system was ready for delivery. This may not be a problem if the prototype meets the users' expectations and the environment can absorb the load of processing without affecting others.

The main argument for allowing the users to exercise the prototype is that these users will discover the problems in procedures and unacceptable system behavior very quickly. An organization may elect to place prototypes in the hands of the users, but before this decision is made, the environment should be studied carefully.

As a minimum, the prototype should be demonstrated before a representative group of users. This demonstration should consist of a detailed description of the system operation, data entry, report generation, and procedure execution. The system structure should be described in detail. Above all, the users must be made to understand that the prototype is not the final product, that it is flexible, and that it is being demonstrated to find glaring or subtle errors from the users' perspectives.

The result of these demonstrations will include requests for changes, corrections to errors, and overall suggestions for enhancing the operation of the system. Once the demonstrations have been held, the prototyping team will reiterate latter steps in the prototype development process to develop the changes, corrections, and enhancements deemed necessary through consensus of the prototyping team, the end users, and management.

For each iteration through prototype development, demonstrations should be held to show how the system has changed due to specific feedback from users and management. This technique should help to eliminate the "we versus they" syndrome found in classical development life cycles which stems from separation of the development team and the users. The demonstrations increase the users' sense of ownership, especially when they can see direct effects from suggestions made during the prototype development phase. Above all, the changes must be developed quickly and demonstrated promptly to the users to get the maximum effect from feedback.

The results of demonstrations will directly influence the number of iterations necessary in prototyping before final specifications can be developed. Requirements uncovered in demonstrating and using the prototype may cause profound changes in the system scope and purpose, the conceptual model of the system, or the logical data model. Modifications in any of these will have a cascading affect in modifications to succeeding steps. Because these modifications occur in the requirements specification phase rather than in design, code, test, or operational phases, they are much less expensive to implement by one or two orders of magnitude [Boeh81].

5.5. Revise and Finalize Specifications

At this point, the prototype consists of data entry formats, report formats, file formats, a logical database structure, algorithms and procedures, selection menus, system operational flow, and a draft users manual. Using a review list such as that described in figure 4, the prototyping team reviews each component for inconsistencies, ambiguities, and omissions. Corrections are made and the specifications are formally documented.

The deliverables from this phase consist of formal descriptions of the system requirements, listings of the 4GL commands files for each object programmed (i.e., screens, reports, database structures, etc.), sample reports, sample data entry screens, the logical database structure, data dictionary listings, and a risk analysis. (The risk analysis should include the problems and changes that could not be incorporated into the prototype, and the probable impact that they would have on development of the full system and subsequent operation.)

1. Statement of Goals and Objectives
2. Definition of System Scope
3. System Diagrams
4. Object Definitions
5. Data Dictionary Report
6. Risk Analysis
7. Logical Data Model
8. Data Entry Screens
9. Report Layouts
10. Selection Menus/Operational Flow
11. Physical Database Structure
12. Draft Users Manual

(Each of the above elements is indexed and cross-referenced by subject and component to insure that all elements are present, that all components have been defined, and that there are no ambiguities or conflicts.)

Figure 4. Specification Review Highlights

5.6. Develop the Production System

At this point, development can proceed in one of three directions:

- **Suspend or cancel the project** because the prototype has highlighted insurmountable problems, or the environment is not ready to mesh with the proposed system.

- **Throw away the prototype** because it is no longer needed or because it is too inefficient for production or maintenance, and continue development in a third or second generation language using a classical development methodology.

- **Continue iterations** through prototype development, each time adding more system functions and optimizing performance until the prototype evolves into the production system (i.e., evolutionary prototyping).

The decision will generally be based on factors such as the following:

- Actual cost of the prototype
- Problems uncovered during prototype development
- Estimated cost of developing the system in a conventional language
- Availability of maintenance resources
- Availability of software technology in the organization
- Political and organizational pressures
- Amount of satisfaction with the prototype
- Difficulty in changing the prototype into a production system
- Hardware requirements

If management decides to cancel the project based on the risk assessment, then this report cannot add to nor detract from the decision. It can only provide system auditors with the road map and directions used by the prototyping team to get to this point.

If the system is to be developed in a 3GL such as COBOL, C, or Ada, then the user of this report is referred to other references that are specifically designed to assist in system development using other life cycle models and methodologies. Prototyping at this point has accomplished the tasks for which it was designed.

The remainder of this paper describes continuing activities in the third direction: the evolution of the prototype into the production system.

5.7. Release Beta Test System

The evolution of the system never stops until the system is no longer necessary or is replaced by another system. As the team adds functions to the evolving prototype, the prototype eventually is transformed into the full production system. When all functions have been tested at the unit level and have been integrated, the system design is documented in detail, the user manual is revised, a training plan is drawn up, and the system is released in beta test mode. This means that the system is probably ready for production but must undergo a shakedown demonstration to make sure that it is stable under production conditions.

Beta testing puts the system under full production conditions but does not yet allow the existing system to be replaced. In general, testing is integrated and performed throughout prototyping at two levels: the user level and the system level. Both of these types are embodied in the beta test, but much testing is performed in the development of the data model and in iterations where the end users may exercise various stages of the prototype system. Unit and integration testing are recommended procedures after initial prototyping has been completed. The prototype is a straightforward vehicle for testing not only procedures and data structures, but also the specification of requirements and how well they have been communicated by the users and understood by the developers.

All users are trained in operation of the system. If the system has the potential of affecting safety, or there are legal ramifications for the organization, then the system may be run in a strict test mode. This may involve parallel operation next to the current production system, or testing in a secure environment. Beta test usually means that there are no warranties or guarantees made about the system until a period of perhaps thirty or sixty days elapses without system failure under full projected production conditions.

Errors and changes requested during this period may be implemented based on severity of the errors or need for changes. Prototypes implemented in a 4GL tend to be easy to correct where errors are concerned. Changes and enhancements may be more difficult, but adherence to modular construction of the system should limit repercussions for any specific changes.

5.8. Release the Production System

Once the beta test period is complete and the users have accepted the system, the users manual is updated and produced in its final version. Maintenance of the system consists of periodically updating the software and documentation to add new functions, eliminate unneeded functions, tune the system, etc. Each periodic update, or release, of the system will have gone through at least a major portion of the prototyping cycle.

6. SUMMARY AND CONCLUSION

6.1. Summary

Application prototyping using a Fourth Generation Language (4GL) as the specification medium has generated enthusiastic support in the past five years. Some of the reasons for this support are--

- its effectiveness in communicating system requirements between developers and end users;
- flexibility for reflecting changes in a "what if" mode of analysis; and
- ease of validating "true" user requirements.

The advantages of prototyping cited by various users and authors include--

- its manageability, because it is highly visible and accountable;
- its flexibility in making rapid changes to users' requests for new capabilities;
- the ease of validating information requirements (i.e., "What you see is what you get."); and
- its emphasis on active physical models instead of paper specifications and documentation.

Combining a Fourth Generation Language with prototyping helps to overcome some of the perceived disadvantages of prototyping and emphasizes the benefits by providing the tools and control necessary to develop prototypes rapidly and inexpensively. The development model proposed in this report is based on prototyping in a Fourth Generation Language and consists of the following major tasks:

- Define purpose and scope of the system
- Develop system conceptual model
- Develop logical data model
- Develop a prototype and demonstrate it
- Revise and finalize specifications
- Develop the production system
- Release beta test system

o Release the production system

o Iterate the cycle (maintain the system)

Part of the difficulty a few software professionals have in accepting prototyping is in not knowing what to prototype. There are many kinds of prototypes. The three major classes are explorative, experimental, and evolutionary prototypes, each with its own particular area of application.

Another view of prototyping accentuates the logical and physical aspects of a system. These prototypes are known as mockups, functional prototypes, and simulations. Mockups address the user interface as the primary target. Functional prototypes concentrate on specific capabilities that are required in the delivery system. Simulations are used to estimate metrics such as response times, database degradation, throughput rates, and other performance characteristics required in the delivery system. In practice, a mixture of mockups and functional prototypes should produce a realistic set of requirements. It would not be unusual to find that all three types of prototyping are used in many cases.

Organizations considering prototyping in a Fourth Generation Language should review the following factors in making a determination:

o **the user community** -- What types of users will have access to the systems?

o **the hardware environment** -- What hardware is needed to implement a 4GL and prototyping?

o **the technical support personnel** -- What training and experience are necessary?

o **the application domain** -- How complex are the applications? How much is known about the problem area?

o **the 4GL** -- How flexible is it? How many resources does it consume? How maintainable are systems written in the 4GL?

6.2. Conclusion

There are no panaceas to completely eradicate the software productivity problem. The hope of finding a solution in the next few decades cannot ameliorate the need for help now. In lieu of a solution, the most that can be accomplished is to face each individual problem area, define it as precisely as possible, analyze it, and develop methods to overcome some of the shortcomings in the current methods of developing applications.

In effect, application prototyping using a Fourth Generation Language is only one approach to one facet of the necessary evolution of software. Many organizations have tried application prototyping. Some have failed, and many have succeeded. There is mounting evidence, however, that prototyping is more than just a fad. The expanding use of Fourth Generation Languages is allowing software developers to realize significant increases in productivity. For the first time, the information industry has tools and techniques available that allow developers to correct errors quickly. Even significant errors are handled rapidly using a 4GL. Using a 4GL and the life cycle proposed in this report may provide a means for organizations to leverage and control the evolution of software in a highly productive environment.

Prototyping using a 4GL has given software developers many new insights into the development process and has allowed users for the first time to actually see the results of analysis based on working models. It forces the developers and the users to communicate with each other. This is what has given them the freedom to express their ideas in systems that actually work and do what they are expected to do.

REFERENCES

[Adam85] Adamski, Lee, "Prototyping Is: Fast, Effective, Practical; Is Not: New, Magical, a Substitute", Computerworld Vol.XIX No.18, May 6, 1985, pp. ID/23-32.

[Albr79] Albrecht, A. J., "Measuring Application Development Productivity", Proceedings of the Guide/Share Application Development Symposium, Monterey, California, October 14-17, 1979, pp. 83-92.

[Appl83] Appleton, Daniel S., "Data-Driven Prototyping", Datamation, November 1983, pp. 259-268.

[Aqui85] Aquino, Cary, and Donna Rund, "Levis Presentation", Proceedings, Data Administration Users Conference, San Francisco, November 24-27, 1985.

[Blat82] Blattner, Meera, and Richard Frobose, "Prototyping and the Life Cycle of Software Systems", ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping, April 19-21, 1982, Columbia, Maryland, Paper #06.

[Blum83] Blum, Bruce I., "Still More About Rapid Prototyping", ACM SIGSOFT Software Engineering Notes Vol. 8, No. 3, July 1983, pp. 9-11.

[Boar84] Boar, Bernard H., **Application Prototyping: A Requirements Definition Strategy for the 80's**, John Wiley and Sons, Inc., 1984.

[Boeh81] Boehm, Barry W., **Software Engineering Economics**, Prentice-Hall, Inc., 1981.

[Boe84a] Boehm, Barry W., Terence E. Gray, and Thomas Seewaldt, "Prototyping vs. Specifying: A Multi-Project Experiment", Proceedings of the 7th International Conference on Software Engineering, March 26-29, 1984, Orlando, Florida, IEEE Computer Society Press, pp. 473-484.

[Boe84b] Boehm, Barry W., "Verifying and Validating Software Requirements and Design Specifications", IEEE Software, Volume 1, Number 1, IEEE Computer Society Press, January 1984, pp. 78-79.

[Bric81] Brickner, Martin F., **IBM Technical Report TR-03.155, Application Development Without Programming - A Relational Data Base Approach**, International Business Machines Corporation, July, 1981.

[Bric83] Brice, L., J. Connell, and D. Shafer, "Using INGRES as a Rapid Prototyping Device During Development of Management Information Applications", Proceedings of the IEEE Symposium of Automated Tools for Software Development, November 1-3, 1983, San Francisco, IEEE Computer Society Press, 1983, pp. 34-43.

[Broo75] Brooks, Fred, **The Mythical Man Month**, Addison Wesley, 1975.

[Chip84] Chipman, Mary Lou, Marco Fiorello, Peg Kay, Patricia Powell, and Monty Snead, **Toward an Improved FIPS Cost-Benefit Methodology, Phase II: Descriptive Models--General Purpose Application Software Development and Maintenance**, National Bureau of Standards Special Publication 500-116, U. S. Department of Commerce, June 1984.

[Conn84] Connell, John, and Linda Brice, "Rapid Prototyping", Datamation, August 15, 1984, pp. 93-100.

[Conn85] Connell, John L., and Linda Brice, "The Impact of Implementing a Rapid Prototype on System Maintenance", Proceedings of the National Computer Conference (NCC) 1985, AFIPS, Chicago, Illinois, 1985.

[Davi82] Davis, Alan M., "Rapid Prototyping Using Executable Requirements Specifications", ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping, April 19-21, 1982, Columbia, Maryland, Paper #09.

[Dear84] Dearnley, P. A., and P. J. Mayhew, "On the Use of Software Development Tools in the Construction of Data Processing System Prototypes", Approaches to Prototyping, Proceedings of a Working Conference on Prototyping, October 1983, Namur, Belgium, edited by R. Budde, K. Kuhlenkamp, L. Mathiassen, H. Zuellighoven, Springer-Verlag, Berlin, 1984, p. 70.

[DeMa79] DeMarco, Tom, **Structured Analysis and System Specification**, Prentice-Hall, Inc., 1979.

[Drap81] Draper, Jesse M., **Costs and Benefits of Database Management: Federal Experience**, National Bureau of Standards Special Publication 500-84, U. S. Department of Commerce, November 1981.

[EDP84] "Special Report - Fourth Generation Languages and Prototyping", EDP Analyzer, Canning Publications, Inc., 1984.

[Fior84] Fiorello, Marco, Peter L. Eirich, and Peg Kay, **Toward an Improved FIPS Cost-Benefit Methodology, Phase I: Descriptive Models--Data Processing Operations**, National Bureau of Standards Special Publication 500-100, January 1983.

[FIPS64] Federal Information Processing Standards Publication 64, **Guidelines for Documentation of Computer Programs and Automated Data Systems for the Initiation Phase**, National Bureau of Standards, U. S. Department of Commerce, August 1, 1979.

[Fish86] Fisher, Gary E., **A Functional Model for Fourth Generation Languages**, National Bureau of Standards Special Publication 500-138, U. S. Department of Commerce, June 1986.

[Floy83] Floyd, Christiane, "A Systematic Look at Prototyping", Approaches to Prototyping, Proceedings of a Working Conference on Prototyping, October 1983, Namur, Belgium, edited by R. Budde, K. Kuhlenkamp, L. Mathiassen, H. Zuellighoven, Springer-Verlag, Berlin, 1984, p. 1.

[Fong85] Fong, Elizabeth, Margaret W. Henderson, David K. Jefferson, and Joan M. Sullivan, **Guide on Logical Database Design**, National Bureau of Standards Special Publication 500-122, U. S. Department of Commerce, February 1985.

[Goma83] Gomaa, Hassan, "The Impact of Rapid Prototyping on Specifying User Requirements", ACM SIGSOFT Software Engineering Notes Vol. 8, No. 2, April 1983, pp. 17-28.

[Hare82] Harel, Elie, and Ephriam McLean, **The Effects of Using A Nonprocedural Computer Language on Programmer Productivity**, University of California, Los Angeles, Graduate School of Management, 1982.

[Heit82] Heitmeyer, C., C. Landwehr, and M. Cornwell, "The Use of Quick Prototypes in the Secure Military Message Systems Project", ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping, April 19-21, 1982, Columbia, Maryland, Paper #16.

[John83] Johnson, James R., "A Prototypical Success Story", Datamation, November 1983, pp. 251-256.

[Jone77] Jones, T. Capers, "Optimizing Program Quality and Programmer Productivity", Proceedings of GUIDE 45, Atlanta, Georgia, November 1977.

[Jone79] Jones, T. Capers, "The Limits of Programming Productivity", **Tutorial--Programming Productivity: Issues for The Eighties**, Second Edition, IEEE Computer Society Press, Washington, DC, 1986, p. 381.

[MacE82] MacEwen, Glenn H., "Specification Prototyping", ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping, April 19-21, 1982, Columbia, Maryland, Paper #24.

[Mart82] Martin, James, **Application Development Without Programmers**, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[McCr82] McCracken, D. D., and M. A. Jackson, "Life-Cycle Concept Considered Harmful", ACM SIGSOFT Software Engineering Notes, April, 1982, pp. 29-32.

[Orr77] Orr, Kenneth, Structured Systems Development, Yourdon Press, Inc., 1977.

[Ridd83] Riddle, William E., "Advancing the State of the Art in Software System Prototyping", Approaches to Prototyping, Proceedings of a Working Conference on Prototyping, October 1983, Namur, Belgium, edited by R. Budde, K. Kuhlenkamp, L. Mathiassen, H. Zuellighoven, Springer-Verlag, Berlin, 1984, p. 19.

[Smit85] Smith, Wayne, "Alternative Approaches for Successful Prototyping", Computerworld Vol. XIX No. 37, September 16, 1985, pp. 17 and 23.

[Spen86] Spencer, Henry, ARPANET message subject "questions from using lint", University of Toronto, May 15, 1986.

[Tayl82] Taylor, Tamara, and Thomas A. Standish, "Initial Thoughts on Rapid Prototyping Techniques", ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping, April 19-21, 1982, Columbia, Maryland, Paper #40.

[Wass83] Wasserman, A. I., "The Unified Support Environment: Tool Support for the User Software Engineering Methodology", Proceedings: SOFTFAIR, A Conference on Software Development Tools, Techniques, and Alternatives, Arlington, Virginia, July 25-28, 1983, IEEE Computer Society Press, Silver Spring, Maryland, 1983, p. 145.

[Wein85] Weinberg, Gerald M., and Daniela Weinberg, "What Do Users Really Want? Part I: The 30-Minute Expert", Journal of Information Systems Management Vol. 2, No. 2, Sprint 1985, pp. 68-71.

[Youn84] Young, T. R., "Superior Prototypes", Datamation, May 15, 1984, pp. 152-158.

[Zave82] Zave, Pamela, "Position Statement", ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping, April 19-21, 1982, Columbia, Maryland, Paper #45.

GLOSSARY

Alpha testing -- The testing of software designed as part of the development process and undertaken as system or operational testing under a controlled/simulated user environment before delivery to the customer.

Beta testing -- The testing of software undertaken after delivery to the customer but before acceptance testing. This phase of testing is considered an operational test under production conditions by agreement of the customer.

Fourth Generation Language -- An automated application development system that provides integrated user functions, data management functions, and system functions at a high level for use by end users and professional data processing personnel [Fish86].

Methodology -- The methods, rules, and procedures that are organized to solve problems in a specific problem area or domain.

Modularity -- The concept of organizing system components into physical and logical groups based on various characteristics, such as function, subsystem, input-output, etc.; the extent to which a system is composed of modules.

Prespecification -- The act of specifying the requirements, design, testing, and evaluation of a software system before all requirements and other factors are known, on the assumption that all "important" factors are identified beforehand.

Evolutionary Prototyping -- A software life cycle based on the development of prototype systems to validate requirements and to expressly evolve into a delivery system.

Prototyping Strategy -- The course of action followed in software development after the prototype has been created. These courses are (1) throw out the prototype and develop the system in a lower level language, (2) enhance the prototype till it evolves into the production system, or (3) cancel the project as infeasible or no longer needed.

Prototyping Team -- A small team of individuals made up of a programmer/analyst who is experienced in software prototyping, and one or two users from the organization that is requesting the development of a system. The purpose of the team is to develop a prototype, test it, and provide recommendations on changes relating to system requirements. All members of the team need to be trained in the use of a Fourth Generation Language.

Software Development Productivity -- The relative capacity of combinations of organization, technique, methodology, and automated tools to produce and maintain software.

Software Life Cycle -- The definition and organization of control phases through which a software system goes in its lifespan. Typically, these are requirements analysis, functional specification, design, code, test, installation, and operation and maintenance.

Software Prototype -- A model or less-than-complete version of a proposed software application that is developed to verify and validate user requirements before major specification work is done.

Throw-away Code -- A method of developing a system based on software prototypes. The prototypes are coded and thrown away when no longer needed, as opposed to evolutionary systems in which the prototype evolves into the final system.

APPENDIX A -- PROTOTYPING EXAMPLE

The following example situation was taken from [Fong85] and was modified to include sections on prototyping.

INTRODUCTION

"A Federal agency is designing a financial management system. None of the application systems offered by software vendors seem to gracefully accommodate the agency's code structure and its cost accounting procedures for its reimbursable divisions. As a matter of fact, although the individuals on the team surveying these packages are each expert in a particular subject area, they lack a good overview of what their agency's requirements are, or should be."

"A primary objective of the design effort is to gain an organizational perspective of the agency's financial data. The logical database design can then be used to develop a system. Prototyping is being considered for possible candidate parts of the system."

"An important consideration in the logical database design project is that the agency's appropriation from Congress constitutes only 63% of the operating budget. Additional income is provided by contracts with other government agencies and the sale of goods and services to the public sector. The financial management system must be able to charge back costs to customers. Another important consideration is that there is an existing payroll system which must interface with the financial management system."

"An example of a reimbursable division is Instrument Fabrication Division, IFD, whose income from services to other government agencies represents 8% of the agency's budget. IFD relies on other divisions within the agency for functions such as procurement and accounting. IFD finances all management and support services by applying a fixed-rate surcharge to the labor base in some of its own units."

"The sample system chosen, 'Agency Financial Management System', is limited in scope, showing some aspects of in-house financial management for a service-oriented agency. Other federal agencies, whose mission is to administer or disburse government funds, would consider this example system a minor subsystem."

Figure A-1 illustrates the basic organizational structure of the agency.

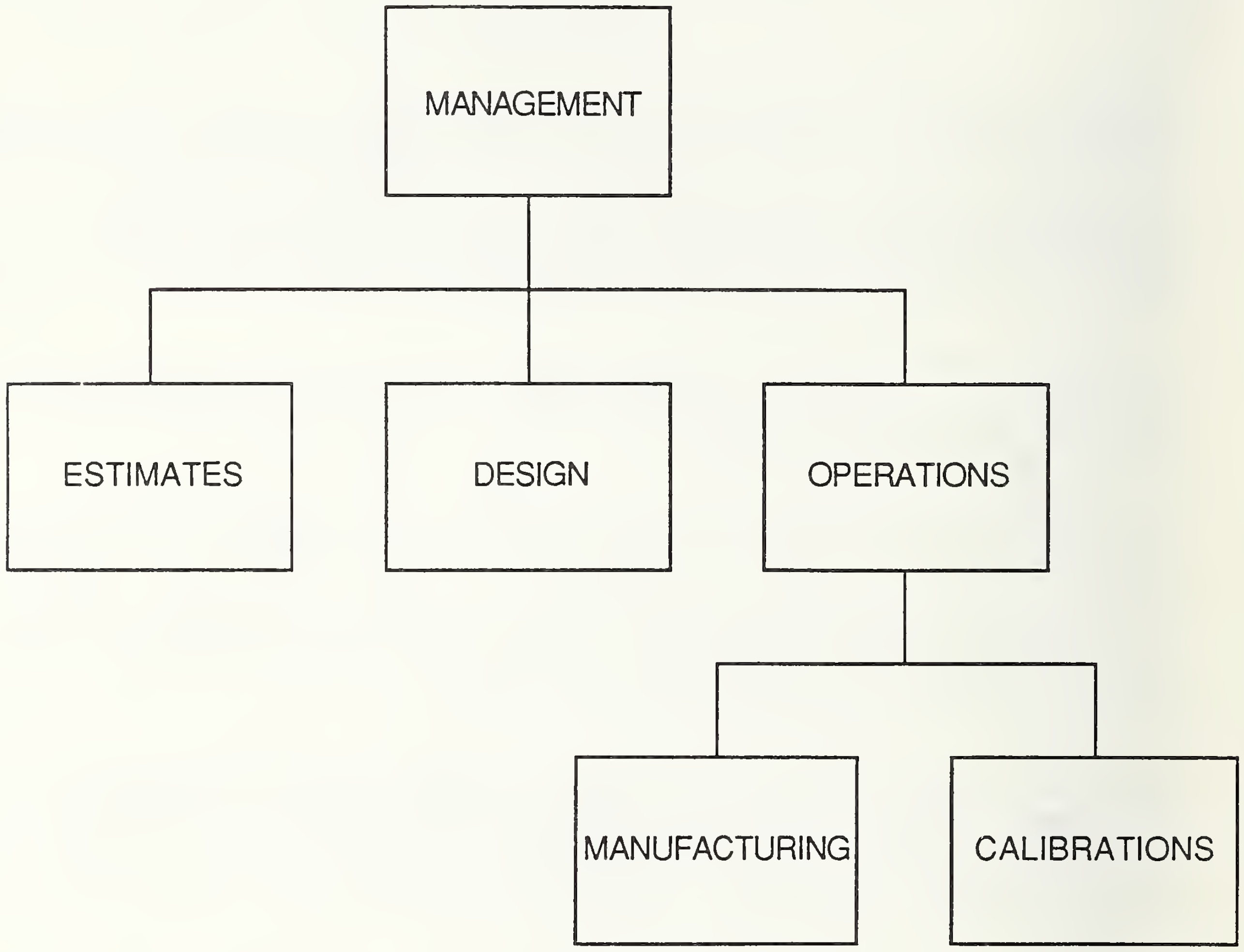


Figure A-1. Organizational Structure of the Agency

A.1. Define Purpose and Scope of System

The tasks associated with defining the purpose and scope of the system include the following:

- o Stating the goals and objectives of the system
- o Defining the system scope by establishing boundaries with other systems and external entities
- o Establishing a team of professional software developers and systems users to define and document a prototype system

A.1.1. Statement of Goals and Objectives

One of the overriding concerns of the effort to develop the financial management system is to gain an organizational perspective of the agency's financial data. Other objectives include incorporating functions such as charging costs back to customers, accounting for various types of operations such as reimbursable funds and procurement, and interfacing with an existing payroll system.

A.1.2. Definition of System Scope

The boundaries of the financial management system are defined as a function of the logical components of the system, the functions to be performed, and the outside world. Specifically, interfaces between the system and the outside world occur between the system and the existing payroll system, the mechanism to allow customers to reimburse the agency for costs, and for vendors to invoice the agency and be paid.

A.1.3. Establish Prototyping Team

The prototyping team is comprised of a professional data processing person who is familiar with the agency's business, expert in the use of a 4GL, and trained in the methodology of prototyping; and one or two prospective system users from the eventual user divisions. In this case, the users would probably come from the accounting and procurement areas of the organization, although the team may be augmented by persons in other highly specialized application areas to provide needed technical expertise from time to time.

A.2. Define System Conceptual Model

The software life cycle is based on development of a conceptual model of the system in question. This model is used to provide a high level conceptual view of the system. Because of its abstract nature, it does not contain the internal detail necessary for implementation. This frees the developers from having to wade through levels of information that may be unnecessary for the required task, such as defining detailed functionality. At a later stage, implementation details may be filled in using the same technique.

Data flow diagramming, one of the techniques proposed for developing the model, is described in [DeMa79]. Data flow diagrams are composed of special symbols that represent the functional flow of information through a system and the conceptual processes that transform the information on its journey. A subset of these symbols is defined in figure A-2. (A subset is used here because the full set contains additional symbols that have meaning primarily in system development rather than in the abstract system concept.)

External entity: A source or recipient of data outside the control of the system. (May be a person, other system, etc.)

Transformation: A manual or automated process that transforms information going into the process into different information emanating from the process. (The terms 'transformation' and 'process' are used interchangeably.)

Data flow: Links that connect system entities (i.e., external entities and transforms) to show the movement of information throughout a system.

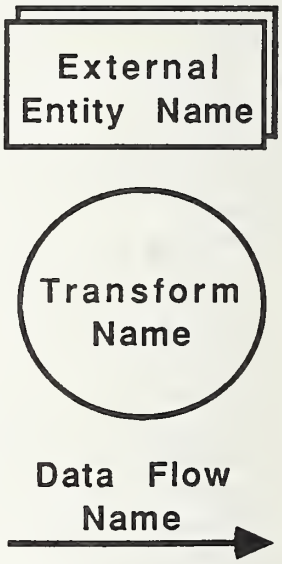


Figure A-2. Data Flow Diagram Symbols

A conceptual view of the system was developed by the prototyping team using these symbols. Figure A-3 illustrates this view.

The diagram shows types of processes rather than names of specific processes (i.e., ACCOUNTING is a type of process, whereas PROCESS-REIMBURSABLE-ACCOUNTS and PROCESS-APPROPRIATED-TECHNICAL-OPERATIONS-ACCOUNTS might be better suited to use in the transformation symbols.) This was done to remove a significant amount of detail in the diagram that would have otherwise obscured the very high level concept of the system. Major information flows are clearly shown as they progress through various interfaces between objects in the system.

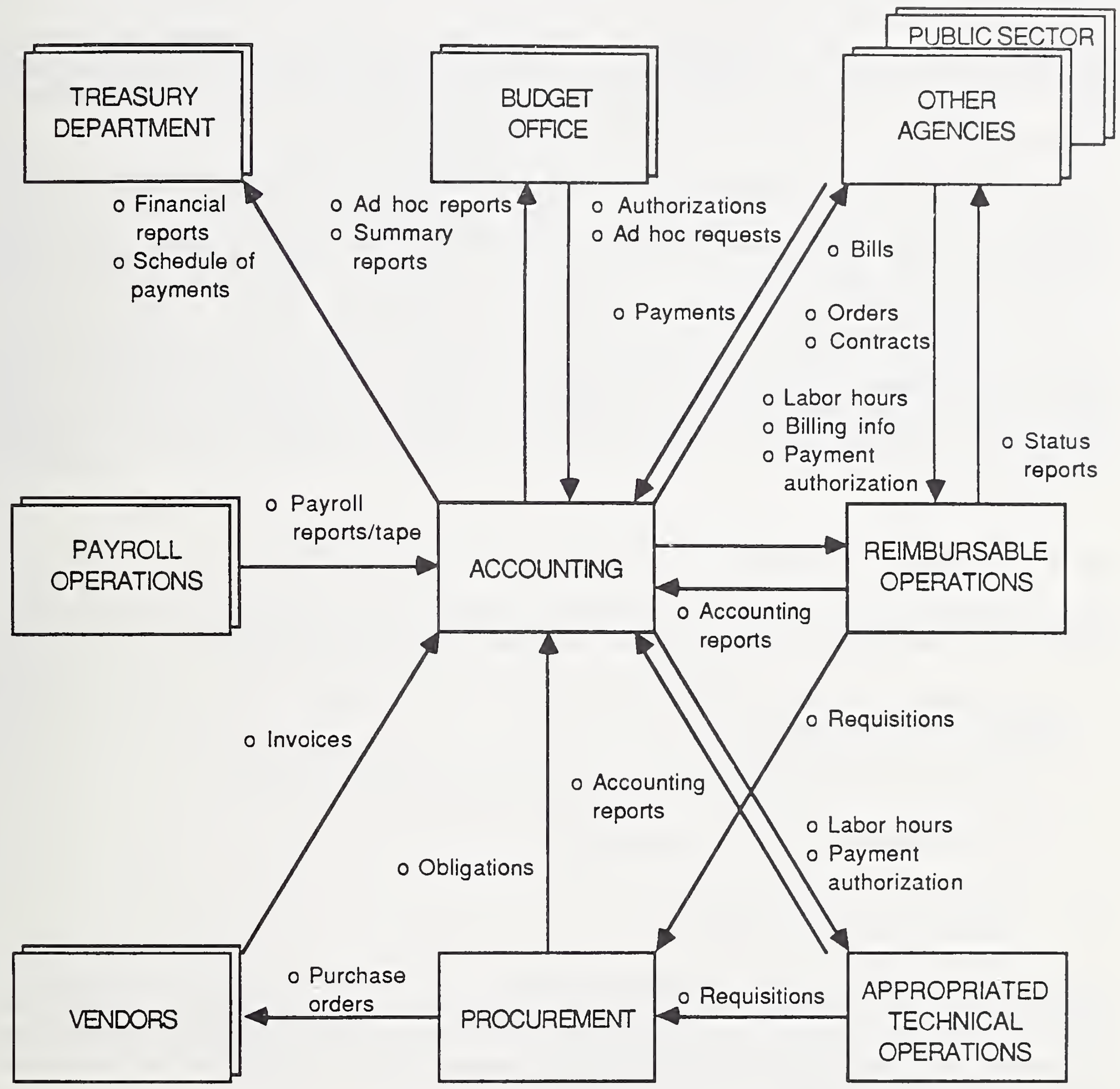


Figure A-3. Conceptual View of System

A.2.1. Identify Major Input and Output

Identifying major input and output of the system is affected by two important factors: 1) the contents and medium of each input and each output, and 2) whether or not the information exists or can be computed from existing information.

One of the most straightforward means of determining these two factors is to analyze the schematic diagrams of the system in its conceptual or functional form. Figure A-3 identifies major input and output as those data flows between external entities and transforms within the system.

Essentially, the task consists of tracing flows of information from external entities down through the system. Each intervening transform and major information flow is assigned a priority based on its distance from the external entity where the path was started. At the conceptual level, this task is generally not complex. (There is usually a highly visible relationship between each object and the requirement which caused its inclusion.) As the conceptual model is decomposed into more and more detailed entities, these relationships become more difficult to judge and tend to drown in the mass of information defined. At the most detailed levels, it helps to look back at this original set of priorities to set the direction for later phases of development.

A.2.2. Determine Feasibility and Estimate Cost and Schedule

Before the team completes the conceptual system model, it prepares an evaluation of the project. The evaluation is based on the following:

- o An estimate of the implementation and life cycle costs and probable schedule of development
- o An estimate of the benefits of the proposed system solution
- o An analysis of the risk in development

A.3. Develop Logical Data Model

Figures A-4 through A-7 represent pieces of the full logical data model developed for this example. Figure A-4 illustrates the high level information flow for the Instrument Fabrication Division. This model shows the types of information passed between organizational components of the division and other components of the agency and the outside world.

Figure A-5, a model of the local information flow between the Estimates Unit and the rest of the world begins to show levels of detail in formation and relationships that were suppressed in the agency and division models. This suppression, or hiding of information, is necessary at higher levels in order to allow the modeler to determine important data relationships at a strategic level and lessen the view-obscuring internal details at the operational level.

This is a major objective of logical data modeling: to see the forest instead of the trees.

Once each component of the division has been modeled, these models are combined into a higher level local information flow model for the division, which is illustrated in figure A-6. The process of abstraction begins to take effect as information flows are synthesized from detailed data structures and reduced to remove redundant information, or ambiguities caused by different names applied to the same entities or attributes, or different meanings ascribed to a name in separate parts of the organization.

In addition, not only are organizational units depicted in the model, but functions that cut across organizational boundaries begin to emerge.

Finally, in the global information flow model illustrated in figure A-7, functional systems and subsystems can be targeted for automation and defined in terms of interfaces, which are in turn defined by the information flows.

The "Boundary of Automation" shown in figure A-7 includes objects such as Accounting and Reimbursable Operations, among others. These objects represent the essential components of the organization, the functions to be performed, and events that will drive the system. They may correspond to systems, subsystems, or specific procedures.

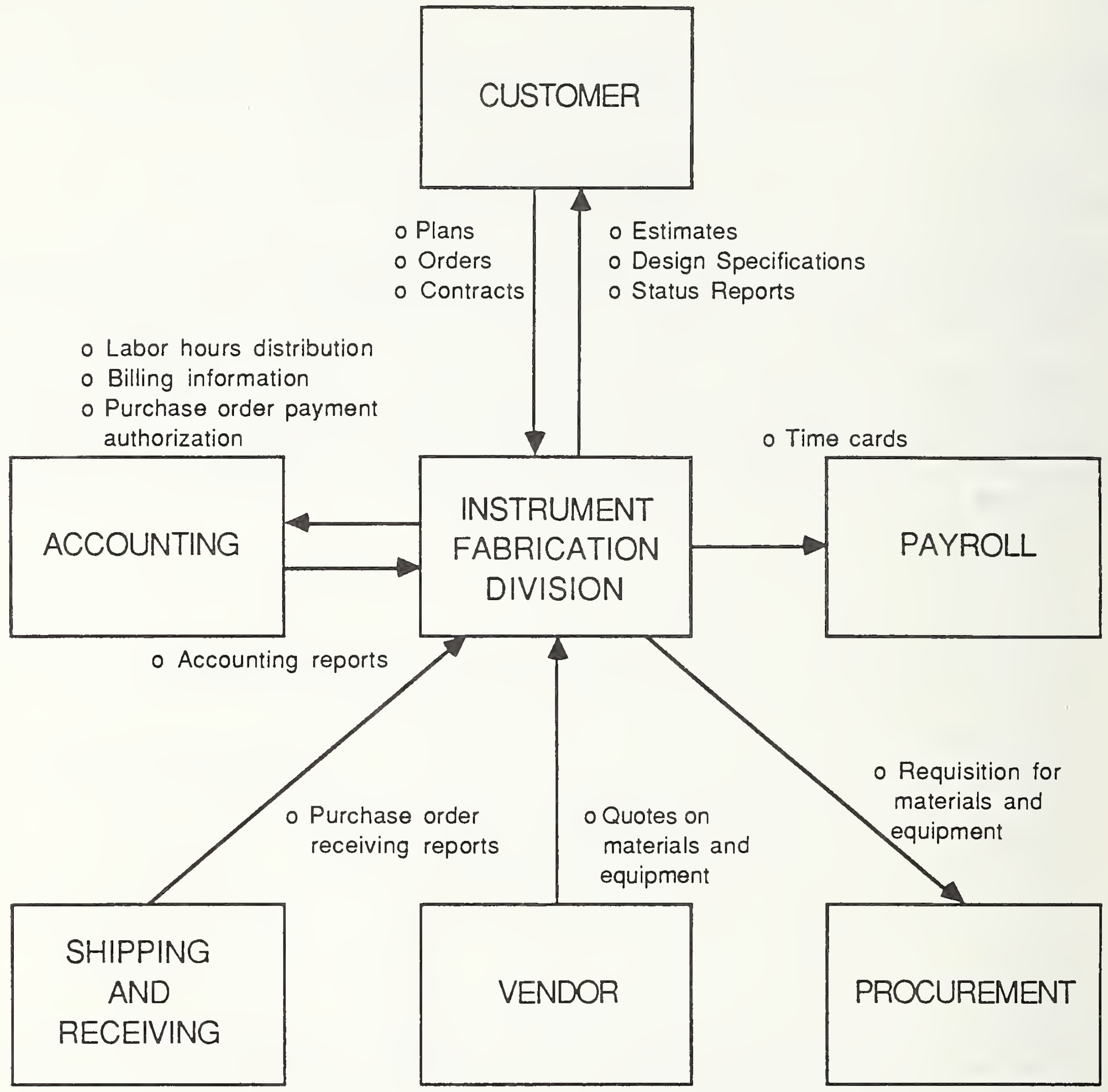

CUSTOMER
o Plans
o Orders
o Contracts
o Estimates
o Design Specifications
o Status Reports
o Labor hours distribution
o Billing information
o Purchase order payment authorization
o Time cards
ACCOUNTING
INSTRUMENT FABRICATION DIVISION
PAYROLL
o Accounting reports
o Requisition for materials and equipment
o Purchase order receiving reports
o Quotes on materials and equipment
SHIPPING AND RECEIVING
VENDOR
PROCUREMENT

Figure A-4. Instrument Fabrication Division

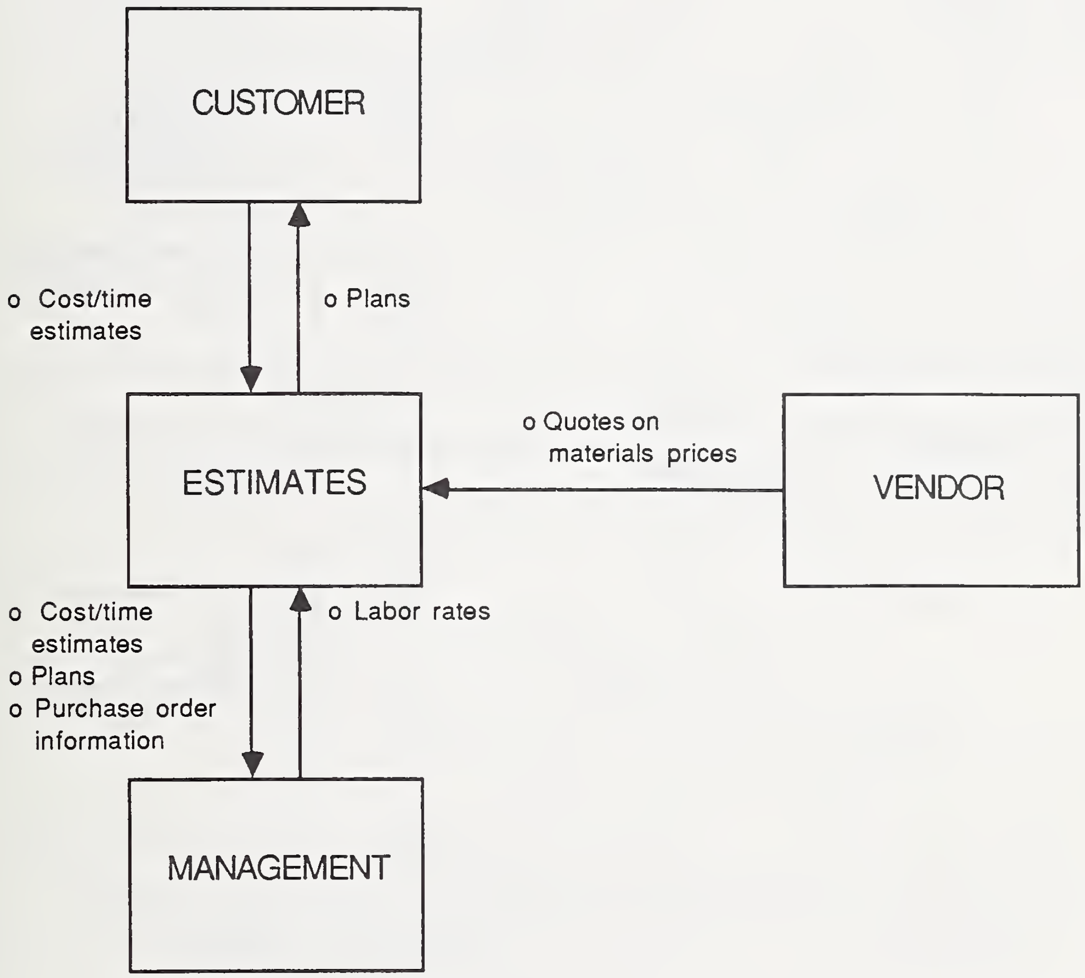


Figure A-5. Estimates Unit Information Flow

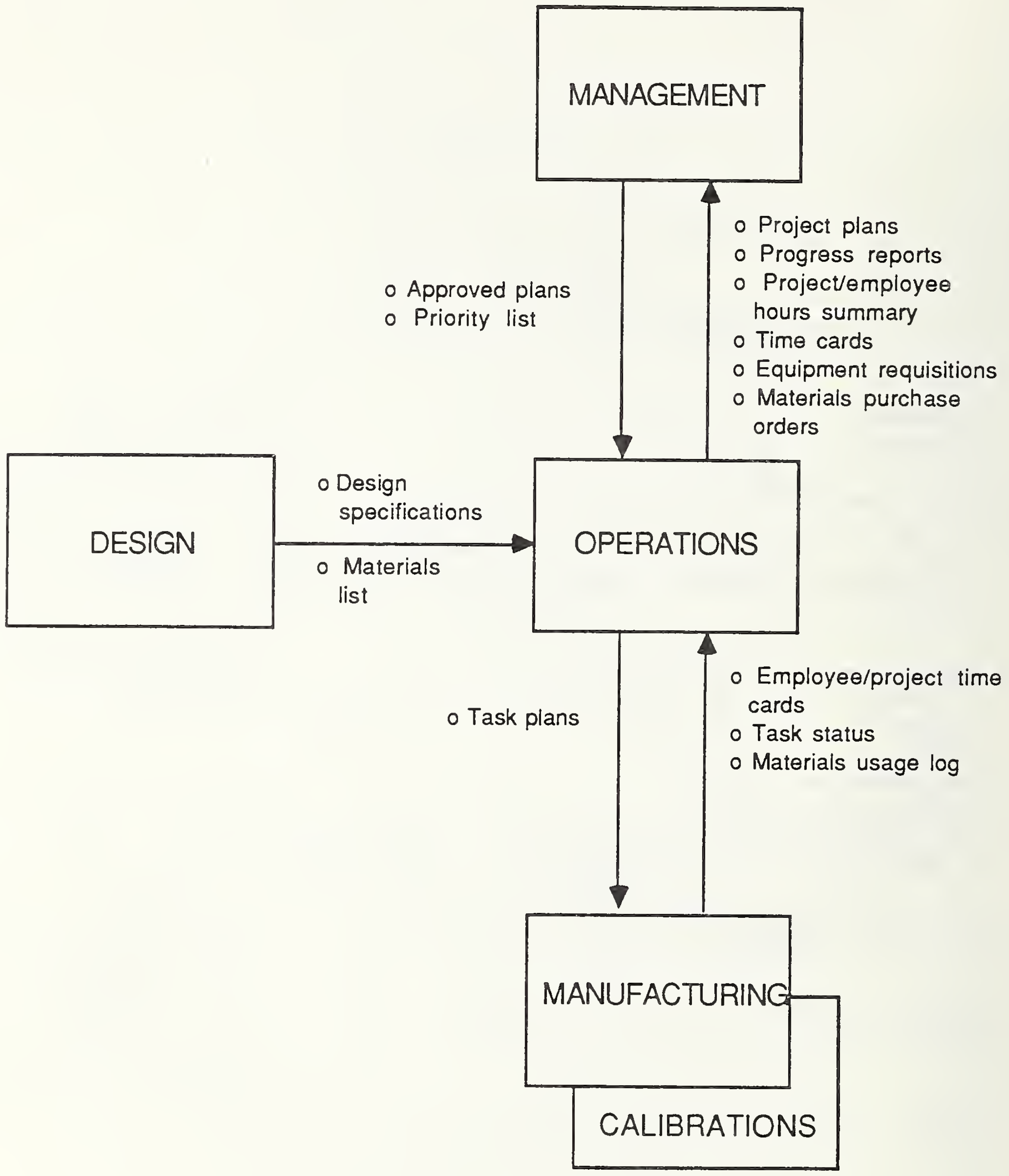


Figure A-6. Division Local Information Flow

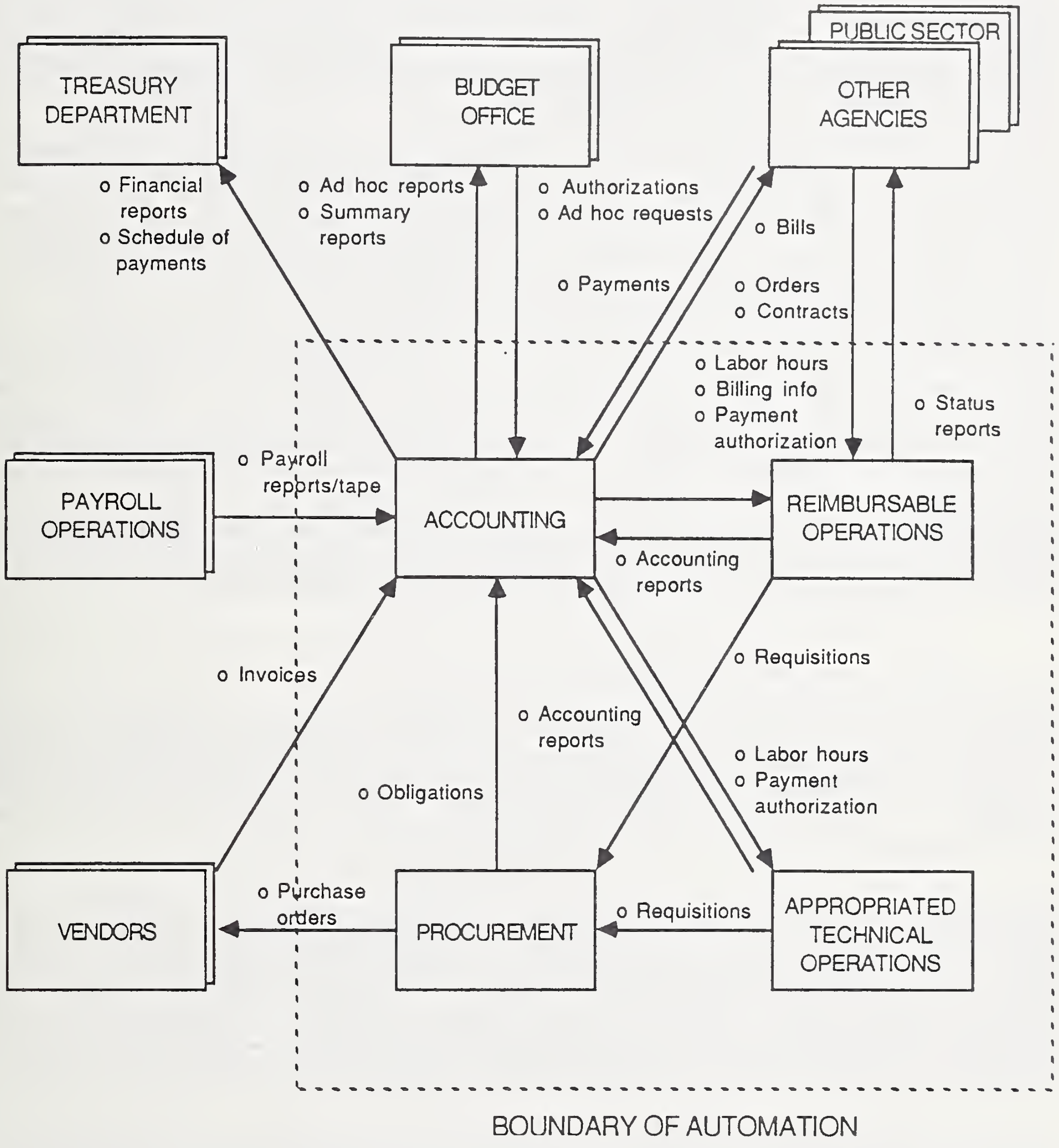


Figure A-7. Global Information Flow Model

A corresponding entity-relationship-attribute data model may then be constructed from information flows and object relationships within a specific section of the global information flow model. Entities become representations of elements such as purchase order line items, vendor identifiers, project time cards, etc. Attributes are composed of individual data elements or fields associated with each entity. Relationships are specified as set constraints where an entity may have a one-to-one, one-to-many, or many-to-many relationship with another entity.

For example, a purchase order may include attributes such as a purchase order number, a work order number, and a vendor identifier. The purchase order may be related to vendors in such a way that one purchase order references only one vendor, but one vendor may be referenced by many purchase orders. In another relationship, one purchase order may be related to one work order, but a work order may generate many purchase orders.

These are the types of concepts that the prototyping team needs to know and have available in order to determine where to concentrate in the development of the prototype. If there are information flows that cross boundaries between possible subsystems or between the system and the external world, these would be likely candidates for inclusion in a prototype. The logical data model provides the structure of the database and a starting point for structuring the system. A prototype is used to tune this structure by refining the procedures for operating the system model and providing realistic data for evaluation of interfaces and function requirements.

A.4. Develop a Prototype and Demonstrate It

Using a 4GL supported by a relational database management system, the prototyping team can create database structures based on the logical data model to support the prototype. Entities becomes records or tables, attributes become columns in those tables, and relationships become indexes and keys depending on the types of entity relationships required.

Data entry and display screens are produced using one or more of several strategies. The data elements found in information flows, entities, or forms used in the organization may be used to provide the structure of the screens. Data entry validation is provided by the relationships defined between entities and by the types of values that specific attributes may contain. Indexes, relational joins, and projections may be used to simulate relationships among entities. Reports can be developed using the same strategies.

Control menus reflect the structure of information flows and follow the basic sequence of procedures that are deemed relevant by the prototyping team. These procedures can be modified through the 4GL to take unforeseen requirements into account.

After all screens, reports, and intermediate procedures have been developed, a draft users manual is created to reflect the operation of the prototype. Test cases are developed and executed to determine if operation is correct and results are as expected. Formal demonstrations can now be made to management and selected users.

A typical demonstration may progress through the following steps:

- o The prototyping team leader describes the features of the demonstration and reiterates that the prototype is only a preliminary and incomplete system. (Note: In some cases, it may be politically expedient to pointedly leave out specific pieces of the system to insure that the prototype cannot be usurped into a production system before it is ready.) The points made in describing the demonstration are--

    - -- that the prototype demonstrates the team's understanding of the system requirements,
    - -- that management should evaluate what is heard and seen in the demonstration to determine if the prototype projects the right concept, and
    - -- discussion of areas that need more emphasis or reevaluation of priorities.

- o A high-level description of the system and the relationship of the prototype to the system are presented.

- o Major flows of information through the system are illustrated using the conceptual model diagram, the logical data model diagrams, and sample screens and reports. These are related to sections in the users manual.

- o Finally, the current project status and planned tasks are reviewed.

Whereas the prototype demonstration to management is primarily informative, the demonstration to users is actually a training technique. The demonstration is designed to illustrate specific tasks and actions on the part of the users and the prototype, and to lead the users through the manual.

A.5. Revise and Finalize Specifications

As user comments and suggestions are received during a trial use period, the prototyping team categorizes them as needing immediate attention or long-range attention. "Immediate" means that the prototype could accommodate a change without disrupting the system concept. "Long-range" means that the change would have affects on the system that could change the system conceptually, and would therefore require more requirements analysis and significant prototyping effort.

The prototype and users manual are updated to reflect immediate changes. In addition, the process of cross-checking all of the documentation is begun. Eventually, the prototype is frozen and final documentation is produced to show the current (frozen) state of the system. The risk analysis is modified to show the projected impact of suggested modifications to the system. A formal system specification is derived from the prototype documentation, and the project schedule and cost estimates are modified to reflect the new requirements.

A.6. Develop the Production System

At the end of the prototyping process, the team should meet with management to discuss future development of the system. The team leader presents the revised schedule, cost estimate, and risk analysis which will serve as the basis for further evolution of the prototype.

The three primary alternatives (i.e., cancel the project, develop in a third generation language, or allow the prototype to evolve into the delivery system) are reiterated and arguments for and against each alternative are presented.

If management decides to continue with evolution of the prototype into the final system, the prototyping team is expanded to include additional analysts, programmers, and technical writers. The team leader makes task assignments based on outstanding changes and planned evolution of the system. Specific tasks may include, but are not limited to, the following:

- All screens and reports are modified to include all data formatting and editing requirements.

- Full-scale testing is performed using the test data developed during prototyping, along with new data and scenarios developed during system evolution.

- Designers document the state of the system for future maintenance.

- Technical editors and trainers prepare training plans using the prototype demonstration scripts and the prototype users' comments and suggestions to guide them.

A.7. Release Beta Test System

Once the tasks of development are complete, users from each division are trained and the system is put online. The development team monitors throughput, system degradation, and problem areas to forecast "mean time to failure" due to software malfunction or hardware overload.

As soon as problems or changes are identified, they are catalogued and scheduled for immediate or future releases of the system.

When the system settles down over a one or two week period without having to halt the system for repairs, a thirty-day benchmark period begins. During this period, if the system has to be halted for repairs, the benchmark period is repeated for thirty days from the last halt. At the end of the thirty day period, the system is prepared for release into production mode.

A.8. Release the Production System

The final version of the users manual is prepared for publication. Examples of screens, reports, and procedures from system operation are merged into the users manual to insure that the instructions do not differ from actual operation.

All software, documentation, and test cases are printed in final form for publication. A magnetic tape copy of this information is made for on-site and off-site storage and backup.

Production software source code and procedures are transferred to the control of the system manager who places these items in the production library.

A.9. Example Summary

The major points brought out in this hypothetical case consist of the following:

- o A prototype provides an inexpensive means of finding out if a system is feasible.
- o Prototyping promotes explicit and continuing interaction among management, users, and developers.
- o A prototype becomes the kernel of a factual set of system requirements.
- o The prototyping team should be small.
- o The team should be knowledgeable in a particular 4GL and software engineering methodology.
- o Management factors, such as risk analysis, schedule, and cost estimates can be monitored and communicated by the prototyping team.
- o All parties (management, users, and developers) must be made aware of the purposes of prototypes.

☆U.S. GOVERNMENT PRINTING OFFICE: 1987 — 181-076/60080

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBS/SP-500/148	2. Performing Organ. Report No.	3. Publication Date May 1987

4. TITLE AND SUBTITLE Computer Science and Technology:

Application Software Prototyping and Fourth Generation Languages

5. AUTHOR(S)

Gary E. Fisher

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)* NATIONAL BUREAU OF STANDARDS DEPARTMENT OF COMMERCE ~~WASHINGTON, D.C. 20234~~ GAITHERSBURG, MD 20899	7. Contract/Grant No.
	8. Type of Report & Period Covered Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Same as item 6

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 87-619824

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

This report describes a methodology for developing software requirements and specifications using Fourth Generation Languages (4GLs) and application prototyping. Various prototyping methodologies are reviewed, and general prototyping strategies and factors are discussed. This report describes the advantages and disadvantages of application prototyping, and develops techniques for implementing a software development model that incorporates prototypes based on the capabilities of 4GLs. The phases, processes, and deliverables are described for each event in the development cycle. An appendix contains a tutorial example to illustrate the methodology proposed.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

4GL; evolutionary software; fourth generation language; prototyping; software lifecycle; software model; software productivity.

13. AVAILABILITY	14. NO. OF PRINTED PAGES 65
[X] Unlimited	
[ ] For Official Distribution. Do Not Release to NTIS	
[X] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.	15. Price
[ ] Order From National Technical Information Service (NTIS), Springfield, VA. 22161	

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, DC 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name ____________________

Company ____________________

Address ____________________

City ____________ State ____________ Zip Code ____________

(Notification key N-503)