SELECTING THE APPROPRIATE APPLICATION DEVELOPMENT METHODOLOGY

By R. N. Burns & A. R. Dennis

This paper synthesizes many of the recent articles about prototyping and contrasts prototyping with the more traditional systems life cycle approach to application development. As some projects are more suited to one methodology than the other, this paper presents a contingency approach, based on project size and project uncertainty, to selecting the most appropriate application development methodology for a given project.

Introduction

A new application system development methodology, prototyping, has recently been discussed in many journals. Information Systems (IS) professionals are no longer restricted to just one approach to developing an application system. One serious deficiency in IS management "is the lack of recognition that different projects require different managerial approaches" [McFarlan]. The purpose of this paper is to meet part of this deficiency by presenting a contingency approach to the selection of an application development methodology.

This paper first summarizes the traditional system development methodology, the system life cycle methodology, and then synthesizes many of the articles discussing a new methodology, prototyping. It analyzes many of the benefits and problems of prototyping to determine for what projects it is most appropriate. Finally, it presents a contingency approach for the selection of a methodology based on project characteristics.

An Overview of the System Life Cycle Methodology

Traditional methodologies, such as the system life cycle methodology, proceed in a linear fashion through several highly structured sequential stages or phases. In the Analysis phase, information about system requirements is gathered and analyzed. In the Design phase, the entire system is constructed on paper, and the user is required to approve the system before the next phase, Program Development, is begun. Once the system has been approved by the user, and is under development, no changes to the system design are generally permitted until the final phase, Implementation, has been completed.

In many cases, this process has resulted in the implementation of unsatisifactory systems, especially for those systems that have a moderate degree of uncertainty regarding exactly what is required by the user [5, 15, 20]. Most of such failures (60-80%), can be traced to an inadequate understanding of user requirements, by the analyst or by the user, when the user approved the initial system design [5]. Many of the remaining failures can be traced to the dynamics of the environment; system requirements changed after the design was approved, but these new requirements were not included in the design because the design was "frozen" after the user approved the system. While "freezing" the design is very efficient for the system developer, it is often ineffective for the user.

R.N. Burns and A.R. Dennis are with the School of Business, Queens University, Kingston, Ontario, Canada K7L-3N6. Telephone (613) 547-6225.

An Overview of the Prototyping Methodology

"The prototyping methodology is based on one simple proposition: users can point to features they don't like about an existing system (or indicate when a feature is missing) more easily than they can describe what they think they would like in an imaginary system" [11]. Rather than force the user to attempt to understand the many minute details of a paper system design specification (often several hundred pages), the developer presents the user with a series of rough approximations (or prototypes) of the computer system. The prototype is not a paper specification of the system, but a working model of the system, albeit often incomplete.

The developer initially meets with the user to collect enough information to build a "rough" initial system prototype, which he then presents to the user for comments. From this concrete approximation of the system, the user is better able to both clarify system requirements, and express those requirements to the developer. The developer then includes the newly expressed requirements in a new prototype, which is again presented to the user for comments. This iterative process is repeated until no new requirements are identified by the user. In essence, the four major phases of the System Life Cycle methodology—analysis, design, program development, and implementation—are combined into one phase, repeated several times [22]. The final system evolves gradually through this process of trial and error, as the user and the developer gradually refine the system.

The prototype is designed with the expectation of change; requests for changes to the system design are accepted, and regarded as a regular part of the development process. The system design is never "frozen" for a long period of time, unable to react to newly identified requirements, or changes in the user's environment. User involvement is a necessary, integral part of the development process.

Due to this high level of user involvement, the user's interest and enthusiasm for the project will have a large impact on the quality of the final system. It is necessary to sustain this interest and enthusiasm throughout the duration of the project. As it is difficult to sustain these for a long time, most prototyping articles emphasize the importance of

short development times—1-2 days to one week at most [20]—even for "relatively large" systems. The need to provide fast development time also requires the use of software tools such as DBMS with sophisticated query language processors, report generators, screen design aids, generalized input processors, and very high level or fourth generation languages [1, 20].

Some Benefits of Prototyping

Many authors have described the benefits discovered by practitioners using the prototyping methodology. A few have formally studied and quantified the benefits. Alavi [1], and Längle, Leitheiser and Nauman [14] have shown that prototyped systems usually result in higher levels of user satisfaction. Prototyping, through its process of gradual refinement, helps match the performance of the evolving system to user expectations, thus increasing satisfaction

Naumann and Jenkins [20], Alavi [3], and Boehm, Gray and Seewalt [6] have also shown that prototyping generally lowers the cost of designing systems. These cost reductions flow from a decrease in time spent on analysis and design, and, to a greater extent, from a decrease in time spent on programming. Programming time is reduced mainly through the use of these sophisticated software tools (fourth generation languages) rather than standard programming languages.

Some Problems of Prototyping

Prototyping often produces less efficient systems than might be developed by a more traditional approach [1, 3, 20], due to this lack of initial analysis and design, and, to some extent, to the less efficient nature of some of the software tools used. (Although Martin [15] has recorded some increases in efficiency when such tools are used, the degree of increase or decrease in efficiency is very much dependent on the particular software tool used.) The solution to these efficiency problems is usually a technical adjustment made to the prototype at the end of the project that is very often less than ideal.

Project management and project control become more difficult with prototyped systems. This is

> "...because the form of the evolving system, the number of revisions to the prototype, and some of the user requirements are unknown at the outset. Lack of explicit planning and control guidelines may bring about a reduction in the discipline needed for proper management (ie. documentation and testing activities may be bypassed or superficially performed). [2]

An Overview of a Mixed Methodology

There is a third development methodology: a mixed methodology that integrates both of the two major methodologies—system life cycle and prototyping. Several authors [3, 4, 5, 8, 9, 10, 15, 20, 21] have touched on the possibility of this integration. Most have addressed the use of prototyping as a strategy to help clarify user requirements during the initial analysis phase of the system life cycle methodology.

> "Prototypes were required in the requirements analysis phase because users could not be sure that computer systems were needed, what functions they should perform, or how they would use them." [10]

> "After a working and satisfactory prototype was developed and the user requirements were clarified, the running prototype provided a basis for writing a set of system specifications. The prototype phase was then followed by the system design and development phases of the life cycle approach." [3]

In these cases, after the user requirements were obtained and the system design was drafted, the prototype was discarded and the specifications were frozen. Prototyping simply replaced the usual analysis phase of the system life cycle methodology.

Another mixed methodology, phased design, has been proposed by Dennis and Burns [8]. With this approach, each major subsystem of the total system is sequentially developed and installed. The entire system is first designed to a coarse level of detail. Then, each major subsystem is selected in turn for refinement. It is designed, presented to the user, refined and approved, before the subsequent subsystems are designed in detail.

A Contingency Approach to Methodology Selection

While prototyping is clearly very suitable for unstructured (or "fuzzy" [1]) systems with a high degree of uncertainty, it is less suitable for large or complex systems [1, 6, 19]. Although other authors have suggested prototyping for large or complex systems [4, 20], it appears as though they are refering to the degree of uncertainty inherent in the project, not to the actual attributes of the system itself.

Most published case examples involving the prototyping methodology deal with the prototyping of interactive information systems [3, 6, 9, 16, 20]. With many systems of this type, "there is relatively little complex logic or processing. There is also a much greater concern with presentation..." [16]. In many of these cases, as the application system simply organizes readily available information into a more comprehensible form. The presentation of the information is more important to the overall success of these application systems than the acquisition of the information in the first place. In short, prototyping is most suitable for systems "requiring more style than substance" [16].

Davis and Olson [7, pp. 563-567] have presented a one-dimensional contingency approach to the selection of a requirements determination strategy. Their approach involves the use of four contingencies (Project Size, Degree of Structuredness, User Task Comprehension, Developer Task Proficiency) to determine the degree of uncertainty inherent in the application system. The greater the uncertainty, the greater the need to use an iterative requirements determination strategy.

A similar approach can be used to determine the most appropriate overall application development methodology. We recommend a two-dimensional contingency approach, based on project uncertainty and project complexity (See Table 1). There are three contingencies to determine *project uncertainty,* based on those proposed by Davis and Olson and McFarlan [7, 17]:

1. Degree of Structuredness. With a well-structured system, the very nature of the task clearly determines the form of the system. The task is clearly defined and quite often rule-based. The system definition is fixed and not subject to change during the life of the development project. A payroll system is a good example of a well-structured system; the system inputs, outputs, and processing are very clear at the start of system development.
2. User Task Comprehension. The degree of understanding the users have about their present tasks and the exact manner in which the application system will be used to support those tasks is important.
3. Developer Task Proficiency. This "is a measure of the specific training and experience brought to the project by the development staff...It is not a measure of ability or potential but rather of directly applicable experience. This contingency indicates the degree of certainty with which the developer will be able to understand requirements accurately and completely, and develop an application to achieve them" [7]. It includes the developers' familiarity with the user's environment, as well as with the tools with which the system will be built.

There are four contingencies to determine *project complexity.:*

1. Project Size. Project size, measured in manhours, is one proxy for the degree of complexity inherent in the project; large projects are usually more complex.
2. Number of Users. As the number of users of a system increases, the complexity of that system increases.
3. Volume of New Information. The greater the volume of new information generated by a system, the more complex it becomes.
4. Complexity of New Information Production. Some systems produce only a small volume of new information, but require a great deal of effort to do so. In addition to the volume of new information, system complexity is determined by the degree of complexity required to produce that new information.

It must be remembered, of course, that prototyping requires the use of software tools, such as fourth generation languages. If the contingency approach recommends the use of prototyping, but no appropriate tools are available in the application system's environment (for example, microcomputers), one has a problem. The solution is either to acquire the required tools, or use a mixed methodology.

Project Complexity	Project Uncertainty: Low	Project Uncertainty: High
High	Prototyping	System Life Cycle
Low	Prototyping	Mixed Methodology

TABLE 1. Project Uncertainty

Generally speaking, for projects that have a high degree of uncertainty, the most appropriate methodology is prototyping. Its iterative nature is more suited to drawing out uncertain user requirements. If the problem addressed by the new computer application is not highly structured, there is much flexibility in the design of the application system. It becomes more difficult and more important to understand the exact requirements for the new system. If the users do not fully understand the exact nature of the task, they will be hard pressed to explain it clearly and concisely to the IS department, because they cannot grasp the flow of the new system—in entirety—from just paper images of the new system. They need to use it, to feel it, to fully understand its workings. The IS area is the only area in which we expect professionals to design and implement complex systems without regular feedback from the ultimate user. Marketing, manufacturing, and research and development all solicit regular user feedback when developing a new product or process, because there are too many unknowns at the start of the project.

Communication of the system needs from the user to the system developer is difficult if the system developer does not understand the user's environment. Users are often inarticulate about their true needs, requiring the system developer to "read between the lines." If the developer has no past experience in the user's area, this becomes very difficult. Prototyping provides a common language for the developer and the user. Rather than attempting to understand and revise a paper image, the user can actually point to one specific part of the working application system and make very clear what he likes or dislikes about it.

For projects that have a high degree of complexity, the most appropriate methodology is the system life cycle methodology, because its structured approach is more appropriate to the increased integration problems faced by more complex systems. The larger the system, the more complex the information requirements of the user and the more difficult it becomes to determine the user's requirements quickly. More analysis and study is required by the user to identify his true needs. The developer cannot simply rush out and design the system in 1-2 days.

The larger the project, the more difficult it becomes to prototype the entire system. More effort is also required by the developer to implement the user's needs. As the developer can not respond quickly, easily, or inexpensively to evolving user requirements, it becomes important to minimize the changes after system construction has begun—in direct conflict with the goals of prototyping.

As project size increases, managing the system development process becomes more difficult and communication among system developer teams becomes critical. It is extremely important to evaluate the effects of a change in one sub-section of the system on the operation of the entire system, rather than rushing out and seeing what happens. Does this improvement in one area cause greater problems in another?

Prototyping for one user is not difficult; prototyping for many is. Meeting the often conflicting demands of several users requires a more careful, controlled approach than the trial and error approach used by prototyping. A modification requested by one user should be approved by all those affected; change is no longer simple.

Most present prototyped systems rearrange existing information. The emphasis is on presenting existing information in a clear, useful style, not on acquiring new information. The creation of new information requires more initial study and planning than that permitted by prototyping. This planning ensures that the information provided by the system is truly effective for the user, and that the information is created in the most efficient manner possible. The greater the volume of new information generated by a system, the more study that must be undertaken to ensure these goals are met, and therefore, the less suited the application is to prototyping.

Most of the new information created by business applications is created in a relatively simple manner. New information is captured beside existing information. For example, the language preference of customers (eg. English, French, Spanish) is added to the customer form and entered by data entry clerks along with name and address. Or, new information may be generated by simple calculations. For example, the average sales to each customer can be calculated by averaging each invoice. These types of new information add to system complexity only when volumes are great.

However, some application systems produce new information that is very complex to generate. A management science application that schedules production in a manufacturing plant, or determines the optimum number of delivery trucks in a parcel service company is not a trivial application system. Such complex systems require a great amount of detailed initial planning, and cannot be designed in just a few days, as required by prototyping. As well, the computational complexity of these systems does not easily lend itself to using the fourth generation languages and other software tools required by prototyping.

For projects that have a high degree of both uncertainty and complexity, a mixed methodology is more appropriate. A mixed methodology incorporates the best of both system life cycle and prototyping, but is not without some problems. The use of a prototyping phase at the start of the project allows the development team room to study the users' needs thoroughly before beginning to write any program code. After a thorough study and analysis, an initial prototype is built. Users can examine and use this initial prototype, to get a better understanding of how the final system will actually work. This will help them clarify their true needs and express them to the system development team. The developers can quickly implement users requests, as changes are easier with this prototype than they would be with the actual system because it is less important to ensure that the prototype is completely bug-free.

Once the users are satisfied, the prototype is discarded and development of the actual system begins. Project uncertainty has been reduced through the use of this prototyping phase. Integration and management problems associated with system complexity now becomes more important. Using our contigency approach, we see that system life cycle is now the most appropriate methodology.

For projects that have low uncertainty and low complexity, any one of the three methodologies would be appropriate. The choice of methodologies would depend to a lessor extent on project characteristics, and to a greater extent on the characteristics of the environment. What methodology is most familiar to the system development team? What is the degree of trust between the developers and the users? Does the environment provide the software tools required by prototyping? In the absence of strong environmental factors requiring one methodology over the others, we would choose prototyping. Prototyping has been shown to have many benefits over system life cycle, and is faster and easier to manage than a mixed methodology.

Conclusion

The IS environment is becoming richer. The characteristics of projects undertaken by the IS department are becoming more diverse in terms of complexity and uncertainty. The methods available to develop projects are becoming more numerous, with the introduction of prototyping and mixed development strategies. No longer can the IS department apply the same development techniques to all projects, as projects with different characteristics have different development needs. The ability to choose the most appropriate development methodology to fit a particular project is becoming more important. This contingency approach will help system developers fit methodology capabilities to project characteristics, thus resulting in higher quality systems.

References

1. Ackoff, R.L. "Management Misinformation Systems", *Management Science,* Volume 17, Number 4, pp. B177-B-186.

2. Alavi, Maryam. "An Assessment of the Prototyping Approach to Information Systems Development", *Communications of the ACM,* Volume 27, Number 6, June 1984, pp. 556-563.

3. Alavi, Maryam. "The Evolution of Information Systems Development Approach: Some Field Observations", *DATA BASE,* Volume 18, Number 3, Spring 1984, pp. 19-24.

4. Alter, S.L. *Decision Support Systems: Current Practice and Continuing Challenges,* Reading, Massachusetts: Addison-Wesley, (1980).

5. Boar, B.H. *Application Prototyping,* New York, New York: John Wiley & Sons, (1984).

6. Boehm, B.W., Gray T.E., and Sewalt, T. "Prototyping vs. Specifying: A Multi Project Experiment", *Proceed-*

ings of the 7th International Conference on Software Engineering, Los Angeles, California: IEEE.

7. Davis, G.B. and Olson, M.H. *Management Information Systems: Conceptual Foundations, Structure, and Development,* New York: McGraw-Hill Book Company (1985).

8. Dennis, A.R. and Burns, R.N. "Phased Design: a Flexible Approach to Application System Development", Queen's University Working Paper 84-19.

9. Earl, M.J., "Prototype Systems for Accounting, Information and Control," *DATA BASE* , Volume 13, Number 2-3, Winter-Spring 1982, pp. 39-46.

10. Groner, C., Hopwood, M.D., Palley, N.A., and Sibley, W. "Requirements Analysis in Clinical Research Information Processing—A Case Study", *Computer,* Volume 12, Number 9, September 1979, pp. 100-108.

11. Jenkins, A.M. "Prototyping: A Methodology for the design and Development of Applications Systems", *Spectrum,* Volume 2, Number 2, April 1985.

12. Keen, P.G.W. "Decision Support Systems: A Research Perspective", *Decision Support Systems: Issues and Challenges,* Toronto, Ontario: Pergamon Press, (1980), pp. 23-44.

13. Langefors, B. "Analysis of User Needs", *Lecture Notes in Computer Science #65: Information Systems Methodology,* Berlin: Springer-Verlog, (1978).

14. Langle, G.B., Leitheiser, R.L., and Nauman, J.D. "A Survey of Application Systems Prototyping in Industry", MIS Research Center Working Paper 83-13.

15. Martin, J. *Application Development Without Programmers,* Englewood Cliffs, New Jersey: Prentice Hall, Inc. (1982).

16. Mason, R.E.A. and Carey, T.T. "Prototyping Interactive Information Systems", *Communications of the ACM,* Volume 26, Number 5, May 1983, pp. 347-354.

17. McFarlan, F.W. "Portfolio Approach to Information Systems", *Harvard Business Review,* September-October 1981, pp. 142-150.

18. McKeen, J.D. "Successful Development Strategies for Business Application Systems", *MIS Quarterly,* Volume 7, Number 3, September 1983, pp. 47-65.

19. McLean, E.R. and Riesing, T.F. "Installing a Decision Support System: Implications for Research", *Decision Support Systems: Issues and Challenges,* Toronto, Ontario: Pergamon Press, (1980), pp. 127-141.

20. Naumann, J.D. and Jenkins, A.M. "Prototyping: The New Paradignfor Systems Development", *MIS Quarterly,* Volume 6, Number 3, September 1982, pp. 29-44.

21. Naumann, J.D., Jenkins, A.M. and McKeen, J.D. "Determining Information Requirements: A Contingency Method for Selection of a Requirements Assurance Strategy", *The Journal of Systems and Software,* Volume 1, Number 4, 1980, pp. 273-281.

22. Parker, D.C. "The Evolution of Management Decision Support Systems", *Information Systems,* Volume 4, Part 4, 1983, pp. 56-65.