View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

D1-61

P.1N

N87-24906

AN APPROACH FOR ASSESSING SOFTWARE PROTOTYPES

V.E. Church, D.N. Card, W.W. Agresti, and Q.L. Jordan*

ABSTRACT

A procedure for evaluating a software prototype is presented. The need to assess the prototype itself arises from the use of prototyping to demonstrate the feasibility of a design or development strategy. The assessment procedure can also be of use in deciding whether to evolve a prototype into a complete system. The procedure consists of identifying evaluation criteria, defining alternative design approaches, and ranking the alternatives according to the criteria.

INTRODUCTION

A software prototype is a functionally incomplete model of a proposed system, built to demonstrate feasibility or explore potential requirements. Most of the interest in prototypes has focused on their development and their role in the software life-cycle. This article addresses prototype assessment--a topic that is less fully developed. Proptotyping has been used most frequently to gain an understanding of user requirements [Gomaa, Scott 81]. When prototyping is employed for this purpose, its benefits can be compared to those of other activities, such as specifying, as a way of proceeding in the early phases of a software development project. Several articles discuss the advantages and disadvantages of the prototyping activity (e.g., [Alavi 84] or [Boehm et al. 84]). In this article, we consider the evaluation and assessment, not of prototyping, but of the prototype itself. When the software prototype is the object being evaluated, two questions are of interest:

- Is the design concept feasible?
- Is the prototype software an adequate basis for further development?

*The authors are with Computer Sciences Corporation, System Sciences Division, 8728 Colesville Road, Silver Spring, Maryland 20910.

Prototyping requires the expenditure of organizational resources, and the resulting prototype, although not a complete system, does have some functionality. Organizations are not always inclined to throw the prototype away; one person's prototype is another person's system. Several articles recommend that organizations consider evolving the prototype into a completed system (e.g., [Duncan 82] or [Blum 83]). Making that decision is significantly different from deciding on the merits of prototyping because it requires evaluating the prototype itself: is it worth the investment of more resources?

When a prototype is being used to evaluate the feasibility of a particular design or development strategy, the prototype itself also needs to be evaluated (see [Giddings 84] for a discussion of uncertainty in software design). The prototype represents one possible approach to solving a problem. Evaluating the prototype requires the consideration of how well alternative designs or development strategies would have addressed the problem. This article will explain one procedure for assessing software prototypes and show how it was applied in evaluating an actual prototype.

ASSESSMENT PROCEDURE

The procedure for assessing a prototype includes three steps:

1. Defining the assessment criteria
2. Identifying the design alternatives
3. Evaluating the alternatives

Defining the Assessment Criteria

The first step is to review the problem statement and extract a relatively small number of high-level requirements to serve as criteria for assessment. We found that, based on the amount of effort required to treat each class properly, the number of criteria should be on the order of 10. Five is probably a lower limit, and twenty is too many to assess in the timeframe implied by a development project.

The assessment criteria represent the users' view of the problem. Each criterion should include a brief statement of requirement (one or two sentences), a short narrative explanation (written in the users' terminology), and an identifying phrase for use in tables and matrices. The intended audience of the assessment is the user--the requirements formalisms that are intended to support software developers are out of place here.

Identifying the Design Alternatives

The prototype represents only one possible solution to the problem. The assessment procedure requires that alternatives be identified as well, so that the prototype can be assessed in the context of other approaches. The second step, then, is to identify approaches to the problem that might provide alternative solutions. The alternative solutions should be based on approaches that are reasonably well understood. It is helpful if an alternative approach can be linked to specific implementations that are concrete instances of the approach.

The alternative solutions should be as different as possible given the constraints of the problem domain. Examples are the use of

- Data base management systems instead of dedicated software
- Fourth-generation instead of procedural languages
- Interactive instead of batch processing
- Distributed instead of centralized processing

The review of such disparate alternatives will certainly increase the confidence level of the assessment and is likely as well to provide useful insights to the eventual development process. The prototype provides a sort of "depth-first" perspective; the examination of alternatives provides the complementary "breadth-first" review.

As with the assessment criteria, the audience for the descriptions of the alternatives is the user. The alternatives should be couched in the users' terminology and presented in narrative form (perhaps a page or two of description per alternative). The number of alternatives will probably be quite small; given the interest in diversity, three to six alternatives will probably exhaust the spectrum of possibilities. Variations on a theme (say, different languages with central versus distributed processing) may increase this to the 5-to-20 range noted above.

Evaluating the Alternatives

Once the sets of criteria and alternatives have been established, the work of judging relative merit can begin. The approach we found most effective was offline individual review leading up to group discussions at which a consensus

evaluation was formed. We found that it was essential to have several different views; no single outlook or experience base could have provided the completeness of evaluation that we sought.

The assessment step consists of ranking the alternatives in order of the degree to which they satisfy each criterion. The essence of the procedure is comparative evaluation of alternatives within each criterion--none of the assessment is performed in a vacuum. Because the prototype is available for inspection and the alternatives are well understood, consensus is easily achieved.

The basis for assigning scores, of course, is the relative value provided to the user--how well is the underlying criterion addressed by each approach? The outcome thereby represents an evaluation of different design concepts tailored very specifically to the problem domain of study. The result of the assessment is a profile of how well the prototype compares with less experimental approaches in the areas of greatest concern. This information provides the basis for decisions on developing the full system.

CASE STUDY--FLIGHT DYNAMICS ANALYSIS SYSTEM

The Flight Dynamics Analysis System (FDAS) is a user-oriented research tool, still under development, that is intended to support spacecraft mission analysts. FDAS will provide computational assistance in planning mission profiles, examining various computational strategies, and performing related flight dynamics ground support activities. It will largely replace a collection of single-use tools and old, much-modified mission analysis programs. Its primary goal (as a new development) is to provide a degree of separation between the analysts (who are generally not particularly avid programmers) and the rather complex support software they require. FDAS is to provide a new, user-friendly approach to performing an existing arduous and error-prone task.

The functional requirements for FDAS were extracted from the existing environment. The prototyped design strategy, however, employed an innovative "software builder" approach not previously attempted for this problem. The planned system would maintain a library of linkable components and provide for their modification and use [Bassett, Giblon 83]. FDAS would provide an integrated system of functions and controls to simplify the programming requirements for the analysts who would use the system.

The prototyping effort was commissioned by the NASA Goddard Space Flight Center to provide a proof-of-concept demonstration of FDAS and to investigate some possible alternatives in the user-interface area [Sukri, Zelkowitz 83]. The prototype assessment effort described here was part of a larger evaluation effort that included examination of comparable efforts elsewhere and actual use of the prototype by spacecraft analysts. This report focuses only on the assessment of the proposed FDAS design (based on the prototype) performed by members of the Software Engineering Laboratory [Card et al. 82].

Step 1. Defining FDAS Assessment Criteria

From the original requirements definition materials and from discussions with eventual users of the system, we developed a set of seven criteria for assessing the concept and design plan for FDAS. These criteria (Table 1) reflect our understanding of the problem to be solved given the constraints of the environment. It was recognized that the new system had to be useful to the existing analysis staff, had to function on existing computer facilities, and had to be maintained by existing operations personnel. Given that, we identified the criteria described briefly below.

- Minimize requirement for global knowledge of the application software--The user should be able to focus on the particular area of concern (e.g., a particular orbit propagator or integration routine) without having to comprehend all of the housekeeping details of the programming system (data transfer, for example, or assignment of FORTRAN COMMONs).

- Minimize requirement for new system-level knowledge--The existing system required user familiarity with editors, compilers, linkers, and the execute modes of two different computers and operating systems. The new system should attempt to reduce the current load of system awareness and to minimize the need for additional system-level knowledge.

- Maximize application-level flexibility and accessibility--The existing software buried functional routines deep within dedicated systems or combined them inextricably into small once-only tools. FDAS should provide accessibility to source code through functionally organized categories. FDAS should further support low-level modification and test of such routines (for example, numeric precision, or type of integration step size determination).

Table 1. Ranking of Alternatives Against Criteria for FDAS Case Study[a]

ALTERNATIVES	CRITERIA							TOTAL
	KNOWLEDGE OF APPLICATION	LEARNING NEW SUPPORT SYSTEM	FLEXIBILITY	EASE OF MODIFYING APPLICATIONS	EASE OF MODIFYING SYSTEM	LEVEL OF INTEGRATION	EASE OF IMPLEMENTATION	
REDEVELOP EXISTING SOFTWARE								
• FORTRAN	9	1	2.5	9	6	8	1	36.5
• OTHER EXISTING LANGUAGE	8	6	2.5	8	5	8	4	41.5
• SPECIAL-PURPOSE LANGUAGE	7	5	5.5	7	4	8	7	43.5
BUILD COMPREHENSIVE DATA-DRIVEN PROGRAM								
• FORTRAN	2	3	8	2	9	2	2	28.0
• OTHER EXISTING LANGUAGE	2	3	8	2	8	2	5	30.0
• SPECIAL-PURPOSE LANGUAGE	2	3	8	2	7	2	8	32.0
USE SOFTWARE BUILDER (PROTOTYPE)								
• FORTRAN	6	7	2.5	6	3	5	3	32.5
• OTHER EXISTING LANGUAGE	5	9	2.5	5	2	5	6	34.5
• SPECIAL-PURPOSE LANGUAGE	4	8	5.5	4	1	5	9	36.5

9921 (114*) 85

[a] 1 = BEST ALTERNATIVE TO SATISFY CRITERION.

2 = WORST ALTERNATIVE TO SATISFY CRITERION.

NOTE: AVERAGE SCORE AWARDED IN CASE OF TIES.

- Minimize effort for application-level modifications--The analytical function of FDAS requires frequent changes to data and software. The effort required for these changes should be minimized.

- Minimize effort for system-level modifications--It is assumed that a maintenance group would be responsible for the addition of new capabilities (a new model of the magnetosphere, for example); the analyst-users would not perform system-level changes. The requirement is that such major changes, providing new system-level functionality, be performed with minimal effort by the support group.

- Provide support for integration of data, software, and analysis--The existing mode of operation involves modification of the software followed by a number of tests and trials using different data and conditions. FDAS should support the data management function of repeating tests, logging runs, and analyzing or comparing output.

- Minimize implementation difficulty--Different approaches present different levels of technical difficulty and probable cost or risk. These aspects should be minimized.

Step 2. Identifying FDAS Design Alternatives

The assessment group defined two alternative approaches (in addition to the prototype) to providing the functionality required of FDAS. Three different programming language options were also investigated as being applicable to the problem domain.

The first alternative was to redevelop existing software. The essence of this approach is to repackage existing functionality within an improved user interface structure. No executive or data-processing functions would be provided except as already available (graphs and plots, for example, are provided in the existing system). The users would continue to rely on the various operating systems and utilities for support.

The second alternative was to build a comprehensive data-driven program, a multifunction system with behavior controlled by user input (similar to various simulation packages, e.g., [Forman 76]). The program would provide both high- and low-level opportunities to control processing. This approach would (conceivably) completely divorce the user from any programming language by providing a higher order of functionality. It also would open the possibility

of using knowledge-based methods for extension and control of activities.

The third alternative (that embodied by the prototype) was to use a software builder approach. The system would maintain an organized library of functions and procedures and would support linking these elements in diverse and unforeseen combinations. The system would support modification of stored routines (including compilation and linkage) and their execution by way of stored command sequences. This approach is similar to some programmer's workbench concepts; it integrates the normally distinct functions of edit/compile/execute/analyze tools into a harmonious whole [Dolotta, Mashey 82].

Three language options were also investigated by the assessment group: FORTRAN, another existing language, or a special-purpose language. FORTRAN is treated separately because it is the language of most existing software and was used in developing the prototype. The cultural bias toward FORTRAN is very strong in the NASA Goddard environment, especially among analysts (whose backgrounds include more engineering, physics, and astronomy than computer science). Any other existing language (for example, Pascal, Ada, HAL/S) would require substantial redevelopment of existing software; it would have to provide a significant added value to be seriously considered. A special-purpose language could be designed and developed specifically for flight dynamics problems and computations. The FDAS prototype, in fact, included an extension to FORTRAN to support data abstractions and modularization (e.g., as in [Isner 82]). Such a language could be much closer to the natural algorithmic methods that are peculiar to spacecraft flight dynamics, but would require both development and user training.

Combining the seven assessment criteria with the nine alternative approaches (three designs and three language options) produces the evaluation matrix shown in Table 1.

Step 3. Evaluating FDAS Alternatives

The evaluators considered each criterion in Table 1 by ranking how well each alternative addressed that criterion. The complete assessment involved considerably more discussion than is presented here [Card et al. 84]. To illustrate this evaluation step, the discussion and rationale for two of the distinquishing criteria are presented here. The results of the assessment are shown in Table 1.

- Knowledge of Application--It was clear that redeveloped software might be easier to use than the existing collection of tools, but the difficulty of modifying the software would still be high. All of the normal difficulty of preventing side-effects and validating interfaces would still plague the analyst-users. We assessed FORTRAN to be the worst offender in this area and assumed that some other language (Pascal was our model) would provide somewhat tidier modularization. Any new language would have as a design goal the minimization of such problems; we scored it higher accordingly.

A comprehensive data-driven program would not permit the user to have access to code at a level of COMMONs and interfaces and so scored very high on this criterion. The development language for the data-driven system would be transparent to the user, thus the different language options provided no discrimination in analyst-user terms.

The software builder approach specifically attempts to hide implementation details from the user by supporting interfaces, data collection, system building, and execution with its own constructs. This approach was thus rated better than the redevelopment approach but less desirable than the data-driven program approach. The arguments for each language option, as discussed for software redevelopment, are applicable here; the rating is shown in Table 1.

- Ease of system-level extension--A different pattern appeared with this criterion. The predominant sequence of language options--FORTRAN last, other language (e.g., Pascal) better, special language best--holds for each of the design approaches, but the relative rankings of those alternatives is different. The software builder approach was judged most accessible to system-level changes and extensions of functionality, partly because the system itself provides some of the tools and means for its extension. The software builder would provide a structure more amenable to change in the directions expected for flight dynamics than is the case with existing software. The redevelopment effort (as we envisioned it) would not provide such an integrated structure. The comprehensive data-driven program approach, because of its monolithic nature (as seen from the outside) would prove most difficult to extend. It should be noted that, in this instance, the language option did affect the ratings in the data-driven approach. The implementation language would have an effect on the ease of programming, as the ratings reflect.

For convenience, we provided a total column in Table 1 to summarize the evaluation across all of the evaluation criteria. In practice, such a total is an oversimplification of the analysis. The final assessment comes from assigning relative weights to each of the criteria and producing a weighted sum. This weighting enables the users' priorities to be reflected in the analysis results.

FDAS Assessment Summary

On the basis of the evaluation, the assessment team found that the prototype had served its purpose in establishing the feasibility of the overall FDAS goal and of the software builder approach in particular. However, the comparative advantages were not large, and not all elements of the prototype were favorably assessed. Drawing on the discussions of competing strategies, the assessment team also suggested some changes to the design approach.

Partly as a result of this analysis, the project underwent an extended operational specification process, leading to a substantially revised design approach along with a greater understanding of how the system would be used. The project is now well into development.

SUMMARY OF THE ASSESSMENT PROCEDURE

Two aspects of the prototype assessment experience should be highlighted: the importance of identifying alternative solutions and the sensitivity of the analysis to the weighting of individual criteria. Evaluating an object in isolation is always difficult, whereas contrasting alternatives is usually easy. By identifying alternative solutions (including the prototyped alternative) to the problem, the evaluators' task is considerably simplified. We found that reaching a consensus evaluation of the prototype was facilitated by the context provided in Table 1. Furthermore, the consideration of alternatives led to recommendations for improving the design approach.

The assessment procedure described here provides a much richer analysis than a simple good/bad evaluation. Not only can the prototype be evaluated in the context of alternative approaches, but the comparative value of different features can also be defined. This procedure approximates competitive development ("flyoffs") more closely than does an acceptance test evaluation, without requiring actual parallel development. This assessment involved a team of evaluators (part-time) over a period of months. The time period coincided with the prototype effort (so no schedule delays were

imposed), but the assessment team was completely separate from, and in addition to, the development team.

As noted above, the assessment procedure supports cost/benefit analysis at a more detailed level than would otherwise be possible. This emphasizes the sensitivity of the procedure to choices of weighting factors for different criteria. Sensitivity analysis can be useful in identifying influential criteria and relatively stable alternatives.

The FDAS experience indicates that this procedure is an effective mechanism for evaluating the feasibility of a prototyped design. It provides an organizing framework for expressing and employing knowledge gained from previous software development experience.

REFERENCES

Alavi, M., "An Assessment of the Prototyping Approach to Information Systems Development," Communications of the ACM, vol. 27, no. 6, June 1984, pp. 556-563

Bassett, P., and J. Giblon, "Computer Aided Programming," Proceedings, SOFTFAIR. New York: Computer Society Press, 1983

Blum, B. I., "Still More About Rapid Prototyping," ACM SIGSOFT Software Engineering Notes, vol. 8, no. 3, July 1983, pp. 9-11

Boehm, B. W., T. E. Gray, and T. Seewaldt, "Prototyping Versus Specifying: A Multiproject Experiment," IEEE Trans. Software Engineering, vol. SE-10, no. 3, May 1984, pp. 290-303

Card, D., et al., The Software Engineering Laboratory, SEL-81-104, NASA/Goddard Space Flight Center, February 1982

Card, D., et al., "A Software Engineering View of the Flight Dynamics Analysis System (FDAS): Parts I and II," Computer Sciences Corporation Technical Memorandum, March 1984

Dolotta, T. A., and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proceedings, Second International Conference on Software Engineering. New York: Computer Society Press, 1982

Duncan, A. G., "Prototyping in Ada: A Case Study," ACM SIGSOFT Software Engineering Notes, vol. 7, no. 5, December 1982, pp. 54-60

Forman, L., "The New York Times Corporate Planning Model," Proceedings of the Winter Simulation Conference. New York: Association for Computing Machinery, 1976

Giddings, R. V., "Accommodating Uncertainty in Software Design," Communications of the ACM, May 1984, pp. 428-434

Gomaa, H., and D. B. H. Scott, "Prototyping as a Tool in the Specification of User Requirements," Proceedings, Fifth International Conference on Software Engineering. New York: Computer Society Press, 1981, pp. 333-342

Isner, J. F., "A FORTRAN Programming Methodology Based on Data Abstraction," Communications of the ACM, vol. 25, no. 10, October 1982, pp. 686-697

Sukri, J., and M. V. Zelkowitz, "Characteristics of Rapid Prototyping Experiment," Proceedings of the Eighth Annual Software Engineering Workshop. NASA Goddard Space Flight Center, November 1983