FEATURE

RAPID PROTOTYPING: LESSONS LEARNED

Opinions on rapid prototyping as a practical development tool vary widely, with conventional wisdom seeing it more as a research topic than a workable method. The authors counter this notion with results from 39 case studies, most of which have used this approach successfully.

V. SCOTT GORDON
Sonoma State University
JAMES M. BIEMAN
Colorado State University

Selecting an appropriate development approach is crucial to building a successful software system. Although the waterfall model remains the most common life-cycle paradigm, interest in evolutionary methods such as rapid prototyping is growing. But is this approach actually being used? If so, how effective is it?

In an attempt to answer these questions, we launched a study four years ago that involved collecting data on as many published case studies of rapid prototyping as we could find and determining commonalities.

In another publication, we documented the effects of rapid prototyping on software quality.[1] Here we extend that work to include not only its effects on the product, but also its influence on the software process itself, in areas like development effort and costing. The study is continuing, and we hope to publish additional results in the near future.

The box on pp. 86-88 describes our evaluation approach. Finding rapid prototyping case studies was not easy. There are many books and research papers on the subject, but few report actual experience. We found only 22 sources of published case studies that include information on the technique's effectiveness. We supplemented these with solicited first-hand accounts, as the box describes. In total, we analyzed the results of 39 studies.

Our overall objective is to use the studies to develop guidelines on how to use rapid prototyping effectively. An important goal is to compare the use of the two main prototyping methodologies: *throwaway*, in which the prototype is discarded and not used in the delivered product, and *evolutionary*, in which all or part of the prototype is retained. In presenting our results, we pay special attention to factors that contribute to the selection of one prototyping method over another.

Our results show that developers considered rapid prototyping a success in 33 of the 39 cases. Of the remaining six, three were described as failures and three did not claim success or failure. We suspect some bias in this number, however, because people are often reluctant to report failures, but even with this bias, the results clearly show that rapid prototyping is a viable development tool. Some of the successful sources address intermediate difficulties encountered and perceived disadvantages.

Another surprise, in light of most attitudes about evolutionary prototyping,[2] was that of the 39 studies, 22 used evolutionary prototyping, while only eight used throwaway (the rest did not state a choice).

The studies reflect military (12), commercial (17), and academic (10) applications. After evaluating the data for common experiences and opinions, we tallied the commonalities to identify recurring themes. We were able to group these into three areas:

♦ *Product attributes.* These include ease of use, match with user needs, performance, design quality, maintainability, and number of features. Although design quality and maintainability are closely related, many studies reported the two separately. In some instances, the authors were able to observe maintenance and so report specifically about it. In other cases, there were maintenance tools.

♦ *Process attributes.* These include effort, degree of end-user participation, cost estimation, and expertise requirements. There were fewer commonalities in this area, perhaps because prototyping itself is a process, and it may not have occurred to authors to report other process effects.

♦ *Problems.* These include performance, end-user misunderstandings, maintenance, delivery of a throwaway prototype, budgeting, and prototype completion and conversion. Authors rarely described actual occurrences of these problems, but many detailed the steps they took to avoid them. As part of our data analysis, we matched stated problems with solutions from other sources.

DEVELOPERS REPORTED THAT RAPID PROTOTYPING WAS A SUCCESS IN 33 OF 39 CASES. MOST USED EVOLUTIONARY.

PRODUCT ATTRIBUTES

Figure 1 shows the six product attributes commonly affected by prototyping. For each attribute, the figure indicates case studies that experienced a positive or negative impact. The box on pp. 86-88 gives the corresponding studies. We also indicate the relative number of studies in which the particular effect was not observed or discussed. The relatively high number of unreported effects reflects the diversity of reporting methods among the case studies.

The first three product attributes — ease of use, user needs, and number of features — can be considered usability factors. The remaining three (performance, design quality, and maintainability) are structural issues.

Usability factors. Users have an opportunity to interact with the prototype, and give direct feedback to designers. Consequently, difficulties with the interface are quickly revealed. Seventeen of our case studies reported improvements in ease of use as a direct result of rapid prototyping; none reported negative effects. After working with a prototype, one user in an academic study soon tired of retyping definitions for each run. This was valuable information for modifying the design.

One commercial developer found that prototyping helped reveal misunderstandings between developers and users. Sometimes users are not sure they want certain functions implemented until they can actually try them, or they may not know they need certain features until actual use exposes an omission or inconvenience. A commercial study found that prototyping helped ensure that the nucleus of the system is right. One academic developer found prototypes useful because "omissions of functions are ... difficult ... to recognize in formal specifications." [Case study 20]

In working with the prototype, users may also find certain features or terminology confusing. Thus prototyping helps ensure that the first implementation (after the prototype) will meet user needs, especially when the prototype includes the user interface. One academic developer states

"The traditional model of software development relied on the assumption that designers could stabilize and freeze the requirements. In practice, however, the design of accurate and stable requirements cannot be completed until users gain some experience with the proposed software system." [Case study 17]

These findings are consistent with Fred Brooks' famous maxim,[3] "plan to throw one away; you will, anyhow."

The effect of prototyping on the number of features in a final system is less clear. Only eight studies reported any data in this area. Thus, the evidence neither supports nor refutes the intuitive notion that prototyping gives the end user a license to demand more and more functionality.

As Figure 1 shows, five cases report that the prototype increased the number of features (in one case, features increased but the author failed to observe the effect). Authors saw the increase as the result of

♦ special-purpose prototyping languages, which made it easy to add new features,

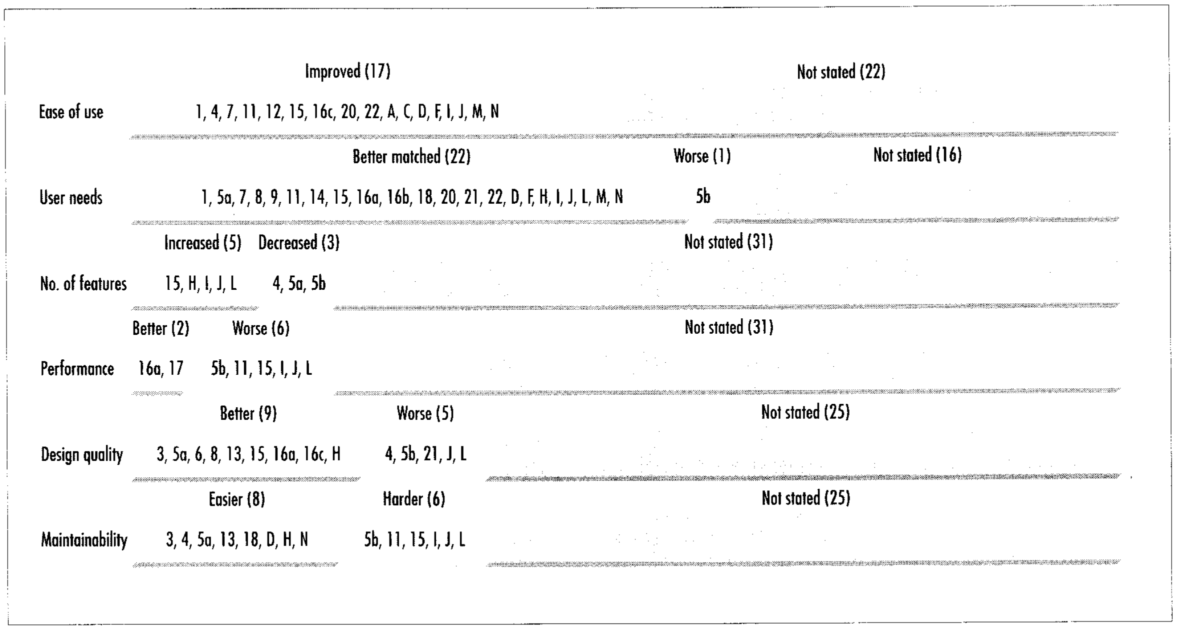


Figure 1. *How case-study developers perceive the effect of rapid prototyping on the product. Referenced case studies appear in the box on pp. 86-88.*

♦ internal software development, which tends to require less time for baselining requirements and thus more time for considering additional features, and

♦ users' demand for more and more functionality, because the prototype lets them visualize increased functionality without considering the cost of adding features.

These authors viewed the increase as a negative effect. One military developer commented "The customer goes crazy adding features and making changes."

In contrast, three studies report a decrease in features as a result of prototyping. In these studies, prototyping caused critical components to be emphasized and noncritical ones to be suppressed, thus reducing the total number of features. One academic study reports that prototyping "fostered a higher threshold for incorporating marginally useful features...." [Case study 4] As a result only the most important features were included in the final product.

One military developer suggested that contractors consider incorporating some mechanism for increasing the cost of the final product when the user requests extra functionality. This might mean costing the prototype phase separately from the rest of development.

Structural factors. In this area, prototyping has more opportunity to produce negative effects. The effect on system performance, for example, depends partly on the prototype's scope. When the prototype focuses solely on the user interface, system performance is likely to be unaffected. When it is used to examine various design alternatives, the outcome can vary.

Sometimes prototyping can lead to better system performance because it is easier to test several design approaches. As one military developer put it, "...with large and complex systems, one's intuition is often a poor guide for identifying the performance-critical regions that will require optimization." [Case study 8] Prototyping can also provide insights into alternative design approaches. One academic developer found that the prototype let the team assess early on the techniques needed to implement specific features.

Evolutionary prototyping, however, can lead to problems when performance is not adequately measured and either inefficient code is retained in the final product or the prototype demonstrates functionality that is unrealizable under normal usage loads. One commercial developer cited "design baggage" as a reason for performance problems. Another academic developer suggested considering performance "as early as possible if [the prototype] is to evolve into the final system." [Case study 11] The case studies contain more evidence of performance problems for evolutionary prototypes than for throwaways.

Design-quality effects are also mixed. More sources indicated an improvement in design quality for both kinds of prototyping, but others report that evolutionary prototyping can result in a less coherent design and more difficult integration. Still others state that the iterative design-modify-review process can yield a better overall design. Three sources reported more lines of code, while four reported fewer lines. However, neither side interpreted this as good or bad.

Quality also suffers when, during evolutionary prototyping, design standards are not enforced in the prototype system. Even when produced with good tools, a design can suffer when developers fail to remove remnants of discarded design alternatives. One commercial developer cited documentation with no design of system procedures and control flows. To avoid these problems, the same devel-

SURVEY SCOPE AND RATIONALE

The 39 case studies included in our analysis describe particular software projects and discuss how prototyping helped or hindered development. In an effort to obtain a good cross-section of practical use, we combined published case studies with more focused surveys of individuals who claimed to have been involved in software-development projects that used rapid prototyping. Consequently, the case studies come from a variety of sources — each with their own motivations and goals — and few addressed all the effects worth tracking.

To obtain some degree of corroboration among the studies, we have tried to locate the effects that were ob-served in at least three case studies. Thus, when reviewing the results by effect, we incurred a preponderance of "not stated" responses. Perhaps we could decrease this number in the future by doing a controlled study, where we knew in advance what effects would be tracked.

The case studies are described below. Each is identified by a number (published) or letter (anonymous), which is used as a reference in the figures in the main text.

Case-study analysis. Case studies varied in degree of rigor. Four sources observed multiple teams (or projects) and presented one set of conclusions based on careful quantitative measurements of the results. Six sources described projects that involved no customer; the goal was simply to create a system for the developers. We avoided drawing strong conclusions about the clarity of requirements or successful analysis of user needs when a project did not involve a separate user.

More often, the cases offer subjective conclusions and suggestions in a less specific, more qualitative manner, acquired from personal experience in a single project. The case studies have varied objectives and intended audiences. One study may report difficulty with a particular rapid-prototyping activity, while another may suggest a remedy for the same problem. Some of the studies include a minimal amount of quantitative measurement interspersed with judgments.

The published sources listed below represent a variety of organizations in academic, military, and commercial environments: AT&T, General Electric, Rand, Mitre, Martin Marietta, Los Alamos National Laboratory, Tektronix, Rome Air Force Base, Hughes, government divisions, and others. The anonymous sources also represent a sampling from these three environments.

Some cases involve separate prototyping and development teams. Ten cases are projects conducted at universities, with only three being student projects. Twelve describe military projects, and the remaining 17 describe other professional software development. (Note that two sources describe multiple projects.)

Attribute selection. The lack of commonalities, the diverse reporting methods, and the varying terminology made attribute selection difficult. We began by identifying attributes (effects) that appeared in three or more sources and used these attributes in our analysis. We then tallied the attributes, along with relevant opinions, observations, and suggestions. We emphasized conclusions that were reached by multiple sources independently.

Although the terminology in the case studies differs, we tried to standardize on the definitions suggested by Bob Patton[1] and B. Ratcliff[2] because we found their terms to be the ones most consistent with the terms used in the case studies. Some terminology remains general because the authors concentrate on different details, and because the software systems described in the case studies are themselves so diverse. "Design quality," for example, can mean many things, depending on the nature of the system or the designer's point of view. One source may illustrate improvement in design quality by specifically listing improvements in code structure, reducing patches, and increasing flexibility, while other sources list different items or none at all.

Although we could have included additional attributes, or subdivided attributes into more specific categories, we did not because the results would have been less clear and there would have been fewer commonalities among the studies. We did not include attributes that the case studies did not mention, even if we thought they might be useful. Also, we omitted some attributes that would have been interesting to observe, such as cohesion, coupling, cyclomatic complexity, and product lifespan, because of insufficient data in the case studies.

PUBLISHED CASE STUDIES

1. *Commercial study.* The author investigates 12 information-systems development projects using the prototyping approach in six organizations. He also presents an experiment involving nine student groups, in which some groups used prototyping and some used specification. The experiment confirmed the findings of the investigation. — M. Alavi, "An Assessment of the Prototyping Approach to Information Systems Development," *Comm. ACM*, June 1984, pp. 556-563.
2. *Commercial study.* The authors describe a methodology for applying object-oriented technology to switching systems and discuss the advantages of using an object-oriented approach in conjunction with prototyping. — E. Arnold and D. Brown, "Object-Oriented Software Technologies Applied to Switching System Architectures and Software-Development Processes," *Proc. Int'l Switching Symp.*, 1990, pp. 97-106.
3. *Military study.* Evolutionary prototyping using object-oriented programming is shown to be an effective way to develop a medium-to-large (50,000 LOC) signal-processing system. — B. Barry, "Prototyping a Real-Time Embedded System in Smalltalk," *Proc. OOPSLA*, ACM Press, New York, 1989, pp. 255-265.
4. *Academic study.* Seven student teams developed the same product: three used prototyping, and four used specification. The authors compare the two techniques. — B. Boehm, T. Gray, and T. Seewaldt, "Prototyping Versus Specifying: A Multiproject Experiment," *IEEE Trans. Software Eng.*, May 1984, pp. 290-302.

5a,b. *Commercial studies.* Both cases involve the use of the Ingres database system. Project 5a involved rewriting a resource-control system for tracking the cost of software projects. It was completed and the system implemented to the apparent satisfaction of all parties.

Project 5b involved developing a requirements document generator. It was criticized by various parties and eventually abandoned. — J. Connell and L. Brice, "The Impact of Implementing a Rapid Prototype on System Maintenance," *Proc. AFIPS*, IEEE CS Press, Los Alamitos, Calif., 1985, pp. 515-524.

6. *Academic study.* The authors describe their experiences applying prototyping techniques to the development of programming languages with advanced features. — R. Ford and C. Marlin, "Implementation Prototypes in the Development of Programming Language Features," *Software Eng. Notes*, Dec. 1982, pp. 61-66.

7. *Commercial study.* The author describes a case study using Promis, a large process-management and information system developed at General Electric to fabricate integrated circuits, and tracks the progress of prototyping and subsequent development. — H. Gomaa, "The Impact of Rapid Prototyping on Specifying User Requirements," *Software Eng. Notes*, Apr. 1983, pp. 17-28.

8. *Military study.* The author describes a large multiphase software- and hardware-development effort in Ada that used incremental development techniques (with rapid prototyping) and object-oriented design. — M. Goyden, "The Software Lifecycle with Ada: A Command and Control Application," *Proc. Tri-Ada*, ACM Press, New York, 1989, pp. 40-55.

9. *Commercial study.* The authors describe how small clinical research programs were developed and evaluated for users who have difficulty clearly expressing their computing needs. Rapid prototyping provided a positive environment, in which users were more willing to participate and better able to express their views. — G. Groner et al., "Requirements Analysis in Clinical Research Information Processing: A Case Study," *Computer*, Sept. 1979, pp. 100-108.

10. *Commercial study.* The author describes 48 Fortune 1000 companies and evaluates the feasibility of discarding prototypes. Although most of the companies use throwaway prototyping, the author concludes that evolutionary prototyping is preferable. — T. Guimaraes, "Prototyping: Orchestrating for Success," *Datamation*, Dec. 1987, pp. 101-106.

11. *Academic study.* The author demonstrates the suitability of rapid prototyping for the development of CAD systems and for object-oriented development. The focus is on the application rather than the merits of prototyping in general. — R. Gupta et al., "An Object-Oriented VLSI CAD Framework: A Case Study in Rapid Prototyping," *Computer*, May 1989, pp. 28-36.

12. *Military study.* The authors track the development of a small prototype messaging system that uses Lisp. Focus is on the role of the prototype, prototyping efforts in general, and reuse of prototype code. — C. Heitmeyer, C. Landwehr, and M. Cornwell. "The Use of Quick Prototypes in the Secure Military Message Systems Project," *Software Eng. Notes*, Dec. 1982, pp. 85-87.

13. *Academic study.* The author describes how rapid-prototyping methodologies were used to develop a prototyping language, EPROL, and a prototyping system, EPROS. — S. Hekmatpour, "Experience with Evolutionary Prototyping in a Large Software Project," *Software Eng. Notes*, Jan. 1987, pp. 38-41.

14. *Military study.* The authors propose a new methodology for rapid prototyping, called software storming, which involves an intense week of videotaped interaction between software designers and users. — P. Jordan et al., "Software Storming: Combining Rapid Prototyping and Knowledge Engineering," *Computer*, May 1989, pp. 39-48.

15. *Academic study.* The authors compare specification and prototyping in a controlled study using student teams. Effort curves were smoother for the prototyping teams, reducing the "deadline effect" common in software development. — W. Junk, P. Oman, and G. Spencer, "Comparing the Effectiveness of Software Development Paradigms: Spiral-Prototyping vs. Specifying," *Proc. Pacific Northwest Software Quality Conf.*, PNSQC, Portland, 1991, pp. 3-18.

16a-c. *Commercial studies.* Describes five cases, three of which are used in the study reported in this article. Project 16a involved an expert system to support a customer-advice service. Project 16b involved a distributed database accessed via a wide-area network. Project 16c involved a real-time control system for quality assurance of a chemical process. The authors attempt to highlight commonalities between the cases, and in the process show that prototyping has advantages over specification. Project 16b can be characterized as large. — H. Lichter et al., "Prototyping in Industrial Software Projects: Bridging the Gap Between Theory and Practice," *Proc. ICSE*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 221-229.

17. *Academic study.* The author introduces CAPS, a computer-aided prototyping system that is essentially an integrated set of computer-aided software tools, and describes its application to the development of a medical system. — Luqi, "Software Evolution through Rapid Prototyping," *Computer*, May 1989, pp. 13-25.

18. *Military study.* Evolutionary prototyping is shown to have advantages over specifying, even on large (100,000 LOC) projects. The authors point out that a significant portion of delivered military systems do not meet original needs and that prototyping can greatly improve this situation. — C. Martin et al., "Team-Based Incremental Acquisition of Large-Scale Unprecedented Systems," *Policy Sciences*, Feb. 1992, pp. 57-75.

19. *Military study.* The author describes a prototyping environment at the Rome Air Development Center, concentrating on a heavily I/O intensive application. There was a significant decrease in development time as a direct result of prototyping. — W. Rzepka, "A Requirements Engineering Testbed: Concept, Status and First Results," in *Proc. Hawaii Conf. System Sciences:*

Vol. II, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 339-347.

20. *Academic study.* The authors describe how retaining prototype code and using a special-purpose prototyping language can be useful for small software projects. — E. Strand and W. Jones, "Prototyping and Small Software Projects," *Software Eng. Notes*, Dec. 1982, pp. 169-170.

21. *Military study.* The author describes how a major ($100 million) military project was successfully implemented using rapid prototyping in a CICS development environment. The study provides insights into how prototyping affects management techniques and the development process in general. — D. Tamanaha, "An Integrated Rapid Prototyping Methodology for Command and Control Systems: Experience and Insight," *Software Eng. Notes*, Dec. 1982, pp. 387-396.

22. *Academic study.* The author describes experiences implementing Backus's FFP System using rapid prototyping. The need for certain functionality became apparent while exercising the prototype. — M. Zelkowitz, "A Case Study in Rapid Prototyping," *Software Practice and Experience*, Dec. 1980, pp. 1037-1042.

ANONYMOUS SOURCES

A. *Academic study.* Employees at a major university describe the development of an online registration system using a special-purpose prototyping environment. They report improved customer satisfaction, ease of use, and ease of training as a direct result of rapid prototyping.

B. *Academic study.* A researcher at a major university reports satisfaction with using Scheme as a prototyping language to produce a working campus-support system in C.

C. *Commercial study.* An engineer at a large telecommunications firm describes problems with rapid prototyping that stemmed from management not understanding the limits of a prototype. The problems resulted in development hardships and premature announcements of delivery dates.

D. *Military study.* An engineer at a large military contractor reports substantially improved product quality, reduced effort, lower maintenance costs, and faster delivery as a direct result of using rapid prototyping. Leveraging with off-the-shelf products helped greatly.

E. *Commercial study.* An engineer at a large data-processing firm reports that prototyping has been quite effective. Recommendations include using an object-oriented approach and throwaway prototyping and carefully selecting an appropriate prototyping language.

F. *Military study.* An engineer at a large military division describes the successful development of small government systems through the use of rapid prototyping. Some results include reuse of typically half the prototype, improved product quality, and better matching with user needs. Some people became upset when their ideas were quickly discredited by experiences with the prototype.

G. *Military study.* An engineer at a small software company reports prototyping worked well in a small project for developing low-level system software. One observation was that you can bid lines-of-code-per-day at a higher rate. Another is that prototyping works only if experienced engineers are available.

H. *Military study.* An engineer at a small military contractor that develops all products using prototyping reports on the use of special prototyping tools that require fewer programming skills to master than typical programming environments.

I. *Commercial study.* An engineer at a large manufacturer reports on the use of evolutionary prototyping to develop very large software systems for workstations. The ability to overlap some of the requirements, design, and coding tasks has improved productivity.

J. *Commercial study.* An engineer at a large communications-and-control contractor describes how proof-of-concept prototypes are developed in Smalltalk, and then the final products are developed using evolutionary prototyping with C. Although total effort has decreased and the product is easier to use, design quality and performance have sometimes suffered because design standards are not rigorous enough.

K. *Military study.* A consultant for a defense contractor reports how rapid prototyping was used for internal support software, but misunderstandings between engineers and marketing staff, along with difficulties in scheduling and costing, have led the company to return to specification.

L. *Commercial study.* An engineer at a large electronics firm describes how operating systems are developed using the Rapid prototyping language. The interface between Rapid and C was the "messiest" part of the final system, but overall the project was successful.

M. *Commercial study.* An engineer at a small software company reports that the organization successfully used rapid prototyping on several nonmilitary internal development projects and that users are much happier with products developed with prototyping methods.

N. *Commercial study.* An engineer at a large communications firm reports that rapid prototyping improved quality, increased user participation, and made final products easier to use. However, developers are often pressured into reusing a throwaway prototype. The engineer recommends carefully defining the prototype's scope and definition.

REFERENCES

1. B. Patton, "Prototyping — A Nomenclature Problem," *Software Eng. Notes*, Apr. 1983, pp. 14-16.
2. B. Ratcliff, "Early and Not-So-Early Prototyping — Rationale and Tool Support," *Proc. Compsac*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 127-134.

oper recommended using a design checklist for each section of incorporated code.

Quality can also be improved by limiting the scope of the prototype to a particular subset (often the user interface), and by including a design phase with each iteration of the prototype. Another option is to completely discard the prototype.

Maintainability effects are similar to those observed for design quality. More studies cite better maintainability than worse. Four described successful maintenance of prototyped systems, even for large projects. (It is with some trepidation that we use the term "large" at all, since it means different things to different people. However, in this article, "large" refers to systems of 100,000-plus lines of code.) The high degree of modularity required for successful evolutionary prototyping can generate easily maintainable code, because such a system is more likely to contain reusable and replaceable functional modules. On the flip side, including hastily designed modules in the final system could cause problems. Indeed, for evolutionary prototyping specifically (not shown in the figure), more sources observe worse maintainability.

Maintenance costs can also go down simply because the final system more accurately reflects user needs. Maintenance often involves correcting invalid requirements or responding to changing requirements. Rapid prototyping can help reduce this type of maintenance because it is likely that user needs will have been better met.

Summary. Overall, we find that software product effects are generally positive, with certain problems related primarily to evolutionary prototyping. We found, however, that developers can use evolutionary prototyping effectively, even for large projects. One academic developer even stated that, for small contracts, throwaway prototyping was economically infeasible compared with evolutionary. Nonetheless, quality attributes such as performance, design quality, and maintainability can suffer during evolutionary prototyping if developers fail to take steps to avoid potential problems, as we describe later.

Unfortunately, the sample size for throwaway prototyping is too small to be statistically useful in evaluating its specific effect on structural factors. (Only two authors of the eight addressed quetions on individual attributes.)

OVERALL, THE EFFECT ON THE PRODUCT APPEARS TO BE POSITIVE, EVEN FOR LARGE PROJECTS.

PROCESS ATTRIBUTES

Figure 2 lists the four process attributes. Again, the figure indicates which case studies observed either a positive or negative effect.

Although the studies discuss other process attributes, such as interface design and language selection, there is less commonality than for product attributes. We do, however, briefly cover these attributes apart from the four in Figure 2 because our overall goal is to describe the effective use of rapid prototyping.

Effort and estimation. One of the most commonly cited benefits of rapid prototyping is that it can decrease development effort. Most of the case studies support this notion. In some instances, the decrease is dramatic. One military study reports a reduction of 70 percent. An academic study reports a reduction of 45 percent. One military project team observed a productivity of 34 lines of code per day, more than six times the estimate for typical military software systems. As one military developer put it, "A buck buys more software now than before." [Case study H]

There are various reasons for this positive effect. Faster design is possible when requirements are clearer or more streamlined. Also, in evolutionary prototyping, part (or all) of the prototype can be leveraged, so the requirements and development efforts tend to overlap.

In one commercial case, development effort increased. The reasons cited were frequent changes in user requirements, which tended to frustrate the designers, and a failure to establish explicit procedures for managing and controlling the prototyping effort. Another commercial developer also suggested the lack of an organized methodology as a possible cause of wasted effort, although it is not clear that this was actually ever observed. In both these cases, prototyping might have decreased development effort if the techniques to manage it had been more effective.

Another commercial study reported an increase in development effort because prototyping revealed that the initial set of requirements was inadequate, and they had to be extended. We think it is unfair to call this a case of increased effort, however, because prototyping actually *prevented* significant wasted effort in the long run.

In viewing cost estimation in a rapid-prototyping environment, the general mood is one of skepticism. One commercial developer observed that the early availability of visible outputs can cause users and managers to be "easily seduced into believing that [subsequent phases] can be skimped on" [Case study 5b] and will be easy to complete. As a result, projects can be underbid. One military consultant cited a case in which a two-year project was sold to management as a six-week project! Most case studies, however, did not address cost estimation in much detail. For that reason, it is difficult to draw any conclusions about the effects of prototyping in this area.

Three sources (two commercial and one military) did offer some suggestions, though. Managers can use rapid prototyping in a contract environment as a separately costed proof-of-concept item. The prototype can be used to test feasibility and/or cost-effectiveness, allowing a customer to abandon a project at a greatly reduced expense. It also gives the developer a chance to better estimate (or abandon) a project that might have been underbid with specification. Developers using proof-of-concept prototyping may have to bid actual development of a complete system separately. Bernard Boar[2] gives a number of suggestions for keeping cost estimates under control.

Human factors and staffing. As Figure 2 shows, most studies reported greater end-user participation in requirements definition. This is not surprising, since users are likely to be more comfortable with a prototype than a specification — which, as one commercial developer noted, can be dull reading. A military source also noted that documentation can be "open to interpretation; [whereas] sample display output is more definitive." [Case study 8] Thus the prototype makes it easier for users to make well-informed suggestions.

As we described earlier, increased user participation has a positive effect on the product by increasing the like lihood that user needs will be met. In fact, lack of sufficient user participation can negate some prototyping benefits. One source describes a case in which the customer's management purposely excluded end users from interacting with the prototype. The intent was to hide the inappropriate allocation of personnel (in a particular division's favor) for as long as possible. Another source observed the same technique, calling it "staff rationalization."

This dangerous political maneuver can be avoided simply by making sure that end users remain actively involved. Because a main advantage of rapid prototyping is in revealing actual requirements, developers should insist on prototype interaction with end users, not just with middle management.

A military developer noted another political difficulty that can surface when using rapid prototyping: "Sometimes our prototype proved that [a particular person's] idea wasn't so great. Result: one upset person/organization." [Case study F]

Eight case studies considered an experienced, well-trained team essential for successful prototyping, because prototyping often involves high-level design decisions about complex programming tasks. Problems can result when inexperienced team members are forced to make such decisions. One case specifically described a project that failed in part because temporary student programmers were thrown into a rapid-prototyping environment. Other case studies indicated that prototyping would not have been successful without highly experienced engineers.

Two cases did use entry-level programmers effectively, attributing their success to the availability of good prototyping tools. However, the failed project that had inexperienced teams was using a fourth-generation-language environment, contradicting the suggestion that having good tools is sufficient; good tools do not guarantee a good system design.

ONE SOURCE REPORTED A DECREASE IN DEVELOPMENT TIME OF 70 PERCENT.

Other effects. One common use of rapid prototyping is to develop a user interface. Various tools exist exclusively for quick development of user-friendly environments, for example Next's Interface Builder and Emultek Corp.'s Rapid. These tools naturally fit into a rapid-prototyping methodology. When special-purpose prototyping tools are used, product maintainability may depend on these tools remaining available.

Early emphasis on the user interface can produce either positive or negative effects, depending on the system being developed.

On the one hand, early development of a prototype gives users the chance to test-drive the software. This aids in clarifying requirements, matching user needs, and creating an easier-to-use product.

On the other hand, there may be a tendency to design the entire system from the user interface. This can be dangerous because the user interface may not characterize the best overall system structure. Thus, a user-interface prototype should be considered part of a requirements specification, not a basis for system design. Again, discarding the prototype is an option, but is only helpful when the prototype's performance is unimportant — *not* the case when the goal is to evaluate the performance of a particular design.

Another attribute that affects prototyping is language selection. Most case studies agreed on the importance of choosing a language suitable for prototyping, but there was much less agreement on what language to choose; in 38 cases, there were 26 languages! The most popular language was Lisp, and it was used in only four cases. Object-oriented methods are receiving increased attention for rapid prototyping;[4] six sources identify object-oriented methods (three used Small-talk) as being particularly well-suited for prototyping. One commercial study even cited the reverse relationship: "...the OO life-cycle almost always in-cludes prototyping." [Case study E]

Two sources cited problems with special-purpose prototyping languages, especially when the language is interpreted rather than compiled. Here, the potential for evolutionary prototyping to result in performance problems is more clear.

PROBLEMS

Most of the studies described successful projects, so there is less direct data on prototyping problems. However, many authors described anticipating and avoiding particular situations. In a few cases, the project could have avoided the reported problem by using a suggestion described in another source. Thus, we were able to see

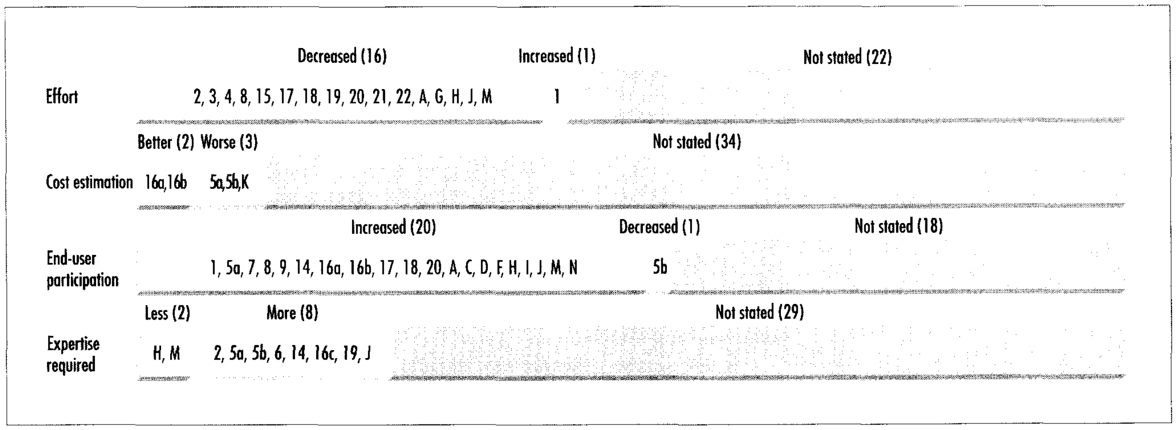


Figure 2. *How case-study developers perceive the effect of rapid prototyping on the development process. Referenced case studies appear in the box on pp. 86-88.*

common problems and match them with possible solutions.

Performance. When prototyping is used to evaluate design alternatives, early measurement of performance is important, and delays in addressing problems can result in design problems that may be costly to repair later.

A prototype can also demonstrate functionality that is not possible under real-time constraints, and this problem may not be discovered until long after the prototype phase is complete. One academic source suggested avoiding this problem by using an open-systems environment, which would make it easier to integrate faster routines when necessary.

End-user misunderstandings. Given too much access to the prototype, end users may equate the incompleteness and imperfections in a prototype with shoddy design. In two cases, this effect contributed to the project's ultimate failure. In another case, rapid prototyping was abandoned as a suitable development method because it gave users the unrealistic expectation that there would be a complete and working system in a short time.

Insufficient knowledge of rapid-prototyping techniques is not limited to users. Sales staff may pass along inappropriate expectations to customers after seeing "working" prototypes. Users then understandably become skeptical or upset when told that development will take longer than they were led to believe. In the studies, high user expectations were typically fueled by too much or too little access to the prototype.

By limiting user interaction to a more controlled setting, developers can keep user expectations at reasonable levels. Users should be clearly told that they are interacting with a mock-up, not a working system, and that the purpose is to clarify requirements. Sometimes developers might want to limit interaction to specific sequences and administer them personally. Further, developers should not oversell the prototype in an effort to impress the customer.

Finally, organizations must train sales and managerial staff to properly understand (and convey) the nature and purpose of the prototyping phase and the prototype itself.

Code maintainability. Certainly, a prototype developed quickly, massaged into the final product, and then hurriedly documented can be very difficult to maintain or enhance. Adding to that is the failure to reevaluate a prototype design before starting to implement the final system. The result is a product that inherits patches from the prototype phase.

To avoid these pitfalls, developers should include documentation criteria in a design checklist, which will help ensure that the prototype is completely documented. Other sources suggested conducting frequent reviews and using object-oriented technology. Discarding the prototype is also an option.

Prototyping can decrease system maintainability when the use of a special-purpose prototyping language results in maintenance engineers having to deal with the prototyping language, the target language, and the interface between them. Complexity can increase, even when system design is good. A prototyped system can become impossible to maintain if it was developed using tools that are not available to maintenance engineers.

Delivering a throwaway. One of the perils of throwaway prototyping is that the prototype may not *get* thrown away. This surprisingly common problem occurs when managers who agree to throwaway prototyping later decide it costs too much to "redo" the system and try to massage the prototype into the product. The resulting system often lacks robustness, is poorly designed, and is not maintainable.

Managers can avoid this problem by maintaining a firm commitment to whatever prototyping paradigm was initially chosen. When a prototype is slated for disposal, then it is best to do so because it was probably not designed with retention in mind. Developers can do their part by fully defining the prototype's scope and purpose.

Budgeting. Three cases described scenarios in which projects were underbid. Because visible outputs are quickly available, managers and salespersons may believe that development is all downhill from here. This breeds overconfidence, which in turn can result in underbidding.

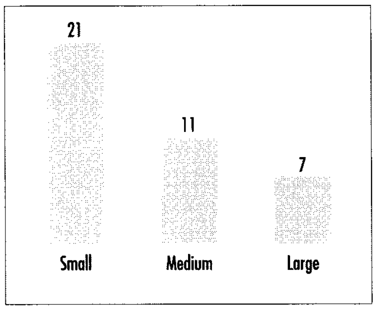


Figure 3. *Distribution of case studies by system size.*

In the case in which the two-year project was presented as a six-week project, management believed that prototyping would achieve fast, working results. Again, organizations must train sales and managerial staff to understand the nature and purpose of the prototyping phase and the prototype itself and to make the distinction between a prototype and a complete system.

Completion and conversion. Prototype development can be time consuming, especially when the purpose and scope of the prototype is not initially well-defined. Boar[2] describes how inadequately narrowing scope can lead to "thrashing" or "aimless wandering" through tasks. Six of the studies support this. Suggested solutions include using a disciplined approach to schedule prototyping activities, carefully defining the prototype's scope, and keeping entry-level programmers out of a rapid-prototyping environment.

Prototyping languages are often used to ease the implementation of a particular system aspect. For example, if the prototype is developed to test user-interface options, the developer will want a language that provides convenient I/O.

However, converting the prototype into the final system may require significant effort and time. This problem is exacerbated when a separate prototyping language is used, especially if the ultimate target language does not have the same simple I/O handling. Another example is the use of an object-oriented language such as Smalltalk with a target language that does not have inheritance.

A few sources observed cost or time overruns. Carefully defining the prototype's scope and systematically comparing the features of both prototyping and target languages can help avoid this problem.

Large systems. As we noted earlier, opinions vary widely on what constitutes a large software system. In one case study, the author described a 200-line system as large! Although some researchers might claim that our definition of large (100,000 LOC) is more suited to the medium range, few would argue that it is small. To distinguish between medium and small projects, we used whatever description the case study reports used and avoided selecting a specific boundary.

Figure 3 shows a breakdown of projects by size. We find no support for the common notion that evolutionary prototyping is dangerous for large projects. In fact, all seven case studies that used evolutionary prototyping with systems of more than 100,000 LOC reported success!

Many case-study authors suggested that problems with evolutionary prototyping will grow in proportion to system size, although the data they report does not provide any direct confirmation of this.

Nonetheless, evolutionary prototyping on large projects can pose problems. It can yield a system filled with patches — hastily designed prototype modules that become the root of later problems. The same problems that challenge performance and maintenance will become more pronounced as the system grows. The same solutions apply, such as using an object-oriented approach or limiting prototyping to user-interface modules, which are less likely to involve important structural design decisions.

Our study identified several surprising prototyping trends and debunked several myths about the approach. For example, we found that rapid prototyping is indeed appropriate for large systems, and there seems to be more successful use of evolutionary prototyping than throwaway. However, with evolutionary prototyping, developers must be careful to address performance issues early in the process, particularly if parts of the prototype are to be included in the final system.

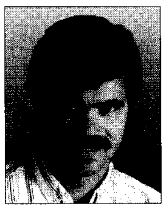
The study also underlined prototyping's potential problems. The most serious are poor design quality and maintainability (especially when using evolutionary prototyping), underbidding, and misunderstandings between developers and users. All of these can be remedied by carefully defining the purpose and scope of the prototype and acting in accordance with its limitations. 91 organizations can further reduce design problems by using experienced, rather than entry-level, programmers whenever prototyping activities involve design decisions. Finally, to avoid creating unrealistic expectations about prototyping, any end-user interaction with the prototype should be carefully monitored.

We recognize that our data is somewhat incomplete. We hope in future work to conduct follow-up surveys of both our published case studies and anonymous sources. We encourage authors of case studies to contact us with updated information. We are also seeking additional case studies, especially descriptions of rapid prototyping in failed software projects.

Our results show that rapid prototyping has had a number of positive effects on both the software product and development process and that it can be used successfully in a variety of situations. We hope that the successes and failures reported here will help those who are now considering this technique. ◆

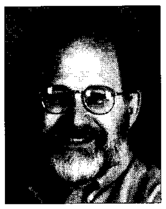
REFERENCES

1. V. Gordon and J. Bieman, "Reported Effects of Rapid Prototyping on Industrial Software Quality," *Software Quality J.*, June 1993, pp. 93-110.
2. B. Boar, *Application Prototyping — A Project Management Perspective*, American Management Assoc. Membership Publications Division, New York, 1985.
3. F. Brooks, *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1975.
4. K. Auer et al., "From Prototype to Product?" *Proc. OOPSLA*, ACM Press, New York, 1989, pp. 482-484.



V. Scott Gordon is a member of the computer and information science faculty at Sonoma State University, where he teaches database systems, programming languages, and computer literacy for educators. As a software developer for TRW Systems, he was involved in the prototyping of an expert system for controlling ground-based mobile emitters. He participated in the work reported in this article while pursuing his PhD at Colorado State University.

Gordon received a BS and an MS from California State University, Sacramento, and a PhD from Colorado State University, all in computer science. He is a member of Upsilon Pi Epsilon.



James M. Bieman is an associate professor of computer science at Colorado State University. He has conducted collaborative research projects with engineers at Storage Technology, Hewlett-Packard, and Martin Marietta, among others. The projects involved the practical application of specifications to software testing and the application of measurement theory to software metrics. He is also interested in the development of metrics for software reuse, cohesion, and object-oriented software.

Bieman received a BS in chemical engineering from Wayne State University, an MPP in public policy from the University of Michigan, and an MS and a PhD, both in computer science, from the University of Southwestern Louisiana. He is a senior member of the IEEE, a member of the ACM, chair of the IEEE-CS Technical Council on Software Engineering's Committee on Quantitative Methods, and chair of the Steering Committee for the IEEE-CS International Symposium on Software Metrics.

Address questions about this article to Gordon at CIS Dept., Sonoma State University, 1801 E. Cotati Ave., Rohnert Park, CA 94928; gordons@sonoma.edu *or* Bieman at CS Dept., Colorado State University, Fort Collins, CO 80523; bieman@cs.colostate.edu