Rapid Prototyping and Software Quality: Lessons From Industry

V. Scott Gordon James M. Bieman

Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096
gordons@cs.colostate.edu, bieman@cs.colostate.edu

Abstract

Empirical data is required to determine the effect of rapid prototyping on software quality. In this paper, we examine 24 published and unpublished case studies, in order to report "real world" experiences. We analyze the case studies to identify common observations, unique events, and opinions. We develop guidelines to help software developers use rapid prototyping in such a manner as to maximize product quality and avoid common pitfalls.

The Authors

- V. Scott Gordon is a Ph.D. student in Computer Science at Colorado State University. He received his B.S. and M.S. degrees in Computer Science at California State University, Sacramento, and has experience as a software developer working for TRW Defense Systems Group. Mr. Gordon's research interests include rapid prototyping, genetic algorithms, and neural networks. He is a member of Upsilon Pi Epsilon and received the 1989 Distinguished Alumni Award from the CSUS School of Engineering and Computer Science.

- James M. Bieman is an Associate Professor in the Computer Science Department at Colorado State University. Dr. Bieman's research is focused on software structure and measurement, testing strategies, software specification and prototyping. He has published in journals including the *Software Engineering Journal*, *Journal of Systems and Software*, and *Computer Languages*, and conferences including COMPSAC and the ACM Testing, Analysis, and Verification Symposium (TAV). He recently served on the Highly Reliable Software Peer Review Committee at NASA Langley Research Center. Dr. Bieman is coordinating the establishment of a Reliable Software Laboratory at Colorado State University.

1 Introduction

Prototyping affords both the engineer and the user a chance to "test drive" software to ensure that it is, in fact, what the user needs. Also, the engineer gains understanding of the technical demands upon, and the consequent feasibility of, a proposed system. *Prototyping* is the process of developing a trial version of a system (a *prototype*) or its components in order to clarify the requirements of the system or to reveal critical design considerations. The use of prototyping may be an effective technique for correcting weaknesses of the traditional "waterfall" software development life cycle by educating the engineers and users [Har87].

Does the use of rapid prototyping techniques really improve the quality of software products? The relationship between development practices and quality must be determined empirically. Our objective is to learn how to improve the quality of software developed via rapid prototyping by drawing on the experiences of documented "real world" cases.

In this paper, we investigate the effect of rapid prototyping on software quality by examining both published and unpublished case studies. These case studies report on the actual use of rapid prototyping in developing military, commercial, and system applications. We analyze the case studies to identify common experiences, unique events, and opinions. We develop some guidelines to help software developers use rapid prototyping in such a manner as to maximize product quality and avoid common pitfalls.

The nomenclature regarding prototyping varies [Pat83]; we use the following definitions: *Rapid prototyping* is prototyping activity which occurs early in the software development life cycle. Since, in this paper, we are only considering early prototyping, we use the terms "prototyping" and "rapid prototyping" interchangeably. *Throw-away* prototyping requires that the prototype be discarded and not used in the delivered product. Conversely, with *keep-it* or *evolutionary* prototyping, all, or part, of the prototype is retained in the final product. The traditional "waterfall" method is also called the *specification approach*. Often prototyping is an iterative process, involving a cyclic multi-stage design/modify/review procedure. This procedure terminates either when sufficient experience has been gained from developing the prototype (in the case of throw-away prototyping), or when the final system is complete (in the case of evolutionary prototyping). Although there is some overlap between rapid prototyping and executable specifications [LB89], we concentrate here solely on rapid prototyping. We generally follow the taxonomy outlined in [Rat88].

This paper is organized as follows. In Section 2 we describe our research methods. Section 3 describes seven effects of prototyping on software quality. Section 4 discusses common beliefs regarding rapid prototyping on quality. We sort out conflicting recommendations concerning four frequently debated questions regarding proper prototyping methods. In Section 5, we describe potential pitfalls associated with prototyping that are revealed by the case studies. We suggest some simple steps to avoid the pitfalls. We summarize our results in Section 6. The References include brief descriptions of each case study as well as general works on prototyping.

2 Nature of Study

For this study, we collected actual case study reports for analysis. Our information is from several available and appropriate sources. These sources include published reports and unpublished communications.

Although many research papers concerning rapid prototyping have been published [LB89, Mit82, Rat88], few papers report on actual real-world experience. We located 17 published reports representing 16 case studies. The earliest case study is from 1979, while most are from

the mid-to-late 1980's (industry use of rapid prototyping appears to be a relatively recent phenomenon). Accounts come from a variety of sources including *Communications of the ACM*, *ACM Software Engineering Notes*, *IEEE Computer*, *Datamation*, *Software Practice and Experience*, *IEEE Transactions on Software Engineering*, and several conference proceedings.

To supplement these published reports, we found additional reports through the internet news service. Through this network search, we have unpublished reports and personal communications from seven individuals closely associated with rapid prototyping. We also include two papers which analyze other rapid prototyping cases. Thus, we have 24 sources of case study information. The sources represent a variety of organizations: AT&T, General Electric, RAND, MITRE, Martin Marietta, Los Alamos, ROME Air Development Center, Hughes, U.S.West, data processing centers, government divisions, and others. Nine of the sources are projects conducted at Universities, but only two of these are student projects. Seven of the sources describe military projects.

The data is not without bias. For example, in 22 of the 24 individual case studies, rapid prototyping is deemed a success. This encouraging result must be tempered by the observation that failures are seldom reported. Some of the sources, however, do address intermediate difficulties encountered and perceived disadvantages of rapid prototyping. Another possible bias occurs in the two sources involving student projects. Boehm describes the inherent bias: "Nothing succeeds like motivation [when] 20 percent of your grade will depend on how much others want to maintain your product" [BGS84]. Finally, four of the sources describe projects which involve no customer *per se*; the goal of these projects is the development of a system to be used by the developers. We do not draw strong conclusions regarding clarity of requirements or successful analysis of user needs when a project does not involve a separate user.

In our analysis, we examine the conclusions made in the case studies with a focus on the impact of rapid prototyping on software quality. Case studies varied in degree of rigor. Three sources observe multiple projects and present conclusions based on quantitative measurements of the results [Ala84, BGS84, CB85]. Others offer subjective conclusions and suggestions acquired from personal experience in a particular project. Some of the studies include a minimal amount of quantitative measurement interspersed with subjective judgement. We emphasize conclusions that were reached by multiple sources independently.

3 Software Quality Effects

The sources describe the effect of prototyping on several aspects of software quality. Most of the described effects are positive.

Improved ease of use

Twelve sources indicate that products developed via prototyping are easier to use. Users have an opportunity to interact with the prototype, and give direct feedback to designers. For example, Gomaa [Gom83] describes how, in some cases, users are not sure that they want certain functions implemented until they actually can try them. Users may also find certain features or terminology confusing. Also, the need for certain features may not be apparent until the system is actually exercised. In Zelkowitz [Zel80], the author "soon tired of retyping in definitions for each ... run", pointing to a need for the capability to store function definitions.

Eleven sources observe more enthusiastic user participation in the early stages of requirements definition. Users are more comfortable reacting to a prototype than reading a "boring" [GS81,

GHPS79] abstract written specification. No sources indicate that software produced via rapid prototyping was more difficult to use.

Better match with user needs

Twelve sources confirm Brooks' famous maxim, "plan to throw one away; you will, anyhow" [Bro75]. Our interpretation is that software developed without a prototype is less likely to meet actual user needs, and be discarded. Sources indicate that prototyping tends to help ensure that the first implementation (after the prototype) will meet users needs: "Omissions of function are often difficult for the user to recognize in formal specifications" [SJ82]. "Prototyping helps ensure that the nucleus of a system is right before the expenditure of resources for development of the entire system" [Ala84]. Only one source indicates that the software produced did not meet users' needs [CB85].

Effect on performance

Three sources suggest that inferior performance is a possible pitfall, especially with evolutionary prototyping.

> "The emphasis in rapid prototyping is typically on proof of concepts rather than performance. However, a programmer should consider performance as early as possible *if the prototype is to evolve into the final system*" [GCG+89].

Developers who intend to use a significant portion of the prototype in the final system need to take steps to ensure adequate system performance (see Section 5).

Effect on design quality

Sources report that keep-it prototyping sometimes results in a system with a less coherent design and more difficult integration. This negative effect can contribute to project failure [CB85] and can impact successful projects [BGS84].

On the other hand, the multi-stage design/modify/review process can result in significantly better overall design. Ford and Marlin state that prototyping "allows early assessment of the efficiency of techniques required to implement specific features" [FM82]. Overall, three sources cite improvement in design quality [Hek87, CB85, FM82], while three sources observe deterioration [BGS84, CB85, Tam82]. See Section 5 for specific recommendations.

Effect on maintainability

Maintainability effects vary. The case studies do not support the belief that keep-it prototyping results in software that is impossible to maintain. Five sources cite improvement [Hek87, BGS84, CB85, A4, Tae91], while only two sources note a reduction in maintainability [GCG+89, CB85]. Hekmatpour describes experiences of maintaining a large system developed via evolutionary prototyping: "The ease with which these modifications were made ... confirms the contention that prototyping can lead to maintainable products" [Hek87]. In analyzing this particular project, we infer that the high degree of modularity required for *successful* evolutionary prototyping can lead to easily maintainable code. Connell and Brice observe that "the modular style of rapid prototype development leads to reusable and replaceable functional modules" [CB85].

There are also indirect reductions in maintenance costs owing to the greater likelihood that user needs will be met *the first time*, reducing the "maintenance" associated with changing requirements [Tae91]. However, some pitfalls should be avoided (see Section 5).

Code length

Three sources report that prototyping results in shorter final programs [AB90, Hek87, BGS84]. No sources report an increase in code length. Boehm's explanation is that prototyping encourages a "higher threshold for incorporating marginally useful features" [BGS84]. Prototyping places a quicker burden of implementation on the designer, and reduces the *talk is cheap* effect of unnecessary promises which are implemented as unnecessary code. Thus prototyping may have a *streamlining* effect. That is, less critical features are less likely to be included in the final system, since the prototype reveals which features are essential. Boehm observed a 40% reduction in source code. Code size may also be reduced when a special-purpose prototyping language is employed [AB90], because features specifically useful for a particular project are typically codified in the language itself and thus do not need to be coded explicitly.

Fewer bells and whistles

None of the sources report an increase in features in the systems due to prototyping. Several sources report discarding some features in favor of others, and three cases specifically cite a reduction in the total number of software features [BGS84, CB85] ([CB85] references two case studies).

This effect is perhaps counter-intuitive. One might expect that the prototyping paradigm gives the end user a license to demand more and more functionality. Actually, Boehm observes that it is more likely that the process will cause critical components to be stressed, and non-critical features to be suppressed [BGS84]. Connell and Brice observe a reduction in features in both successful and unsuccessful cases [CB85].

4 Common Questions

The rapid prototyping literature reveals a number of controversial topics. We describe common questions which are relevant to software quality, and summarize the often conflicting recommendations of our sources.

Should the prototype be kept, or thrown away?

Many engineers are adamantly opposed to keep-it prototyping [A5, Tae91]. Boar's suggestions for rapid prototyping [Boa85] are often cited, and he generally recommends against keep-it prototyping. Guimaraes [Gui87] is more specific in suggesting that the prototype needs to be discarded only if it is used to test complex design alternatives.

Our sources do not support the notion that keep-it prototyping results in poor quality software products. In fact, ten of the studies specifically recommend keep-it (or *evolutionary*) prototyping. Only six authors insist that prototypes be discarded. Three of the case studies [AB90, Hek87, Tam82] represent successful keep-it prototyping on large software projects. On small projects, several sources suggest that keep-it prototyping is essential. Strand and Jones state that "for small-scale systems, clearly the prototype must be a part of the finished system or prototyping is economically infeasible" [SJ82]. Gupta et al. [GCG+89] report that "the environment should encourage code reusability, to extract maximum work from a minimum of code ... to avoid reprogramming as the system evolves." We conclude that both keep-it and throw-away prototyping have pitfalls. See Section 5 for details.

What language should be used to develop the prototype?

Although most sources stress the importance of carefully selecting a language suitable for prototyping, 22 cases employed 18 different languages. One common suggestion is that the language should offer convenient input/output development. Two cases [AB90, GCG+89], suggest object-oriented environments are preferred for evolutionary prototyping. The most popular single language choice was Lisp, although it was used in only three cases [HLC82, Hek87, AB90].

Can prototyping be used for developing large systems?

Of 24 cases, three can be considered large [AB90, Hek87, Tam82], and another eight are medium or medium-to-large [Gom83, BJK+89, CB85, A1, A4, Zel80, GCG+89]. We find no support for the common notion that *evolutionary* prototyping is dangerous for large projects. All three cases involving large projects used evolutionary prototyping. However, the pitfalls involved with evolutionary prototyping seem to grow in proportion to the size of the system being prototyped. See Section 5 for specific recommendations.

Does rapid prototyping require experienced programmers?

Eleven cases used experienced programmers and two used inexperienced programmers. Experience levels are not indicated in the remaining cases. One case utilized novice programmers successfully on a small system in a non-contractual environment [Ala84].

A few of the sources recommend an experienced, well-trained team as essential for successful prototyping. Connell and Brice [CB85] describe a project which failed partly because temporary student programmers were thrown into a rapid prototyping environment. Other sources [A7, Tam82, Ala84] state that experienced (or at least thoroughly trained) engineers are required for successful use of rapid prototyping. Alavi describes a successful small-scale project which utilized entry-level programmers, but then concludes without explanation that "Prerequisites to successful prototyping include ... motivated and knowledgeable users and designers" [Ala84]. Overall, the evidence suggests that it is dangerous to put inexperienced programmers into a rapid prototyping environment.

5 Pitfalls and How to Avoid Them

Much of the literature about rapid prototyping describes inherent pitfalls not found when using the specification approach [Boa85]. Examination of the case studies confirms that these pitfalls are real. Fortunately, the sources also describe similar methods of dealing with each of them.

Inferior design quality

Poor design quality commonly results when a prototype is meant to be thrown away, but is kept instead in order to save costs. Quality also suffers when, during evolutionary prototyping, design standards are not enforced in the prototype system. To avoid this problem, Connell and Brice suggest adhering to a design checklist [CB85]. Code which is transferred to the final product must satisfy the checklist. Quality can also be improved by limiting the scope of the prototype to a particular subset (often the user interface) [Ala84, Tam82, Tae91], and by including a design phase with each iteration of the prototype [Hek87]. Another option is to completely discard the prototype.

Unmaintainable code

A prototype which is developed quickly, massaged into the final product, and then hurriedly documented can be very difficult to maintain or enhance. The advice for avoiding inferior design quality also apply here. Documentation criteria should be included in the design checklist to ensure complete system documentation of the prototype. Other suggestions include frequent reviews [Hek87] and the use of object-oriented technology [GCG+89]. Of course, discarding the prototype is also an option.

Poor performance

The consensus is to build the prototype without initial concern for system performance [Gom83, Zel80, HLC82]. However, the prototype can demonstrate functionality that is not possible under real-time constraints. This problem may not be discovered until long after the prototype phase is complete. One way of avoiding this pitfall is to use an open system development environment to make it easier to integrate faster routines when necessary [GCG+89]. Two sources suggest the early measurement of performance, especially when evaluating design alternatives [CB85, Gui87]. Performance issues are less critical when the prototype focuses solely on the user interface. Again, discarding the prototype is also an option.

A throw-away prototype becomes the product

This common problem (observed by four sources [Gui87, CB85, A3, Tae91]) typically occurs when managers are initially sold on the idea of throw-away prototyping. But, when they see the prototype, managers decide to save money by massaging the prototype into the product. The resulting system often lacks robustness, is poorly designed, and is un-maintainable. [Gui87, CB85, A3, Tae91] stress the importance of avoiding this pitfall; *either plan to keep the prototype, or discard it*. One of the perils of throw-away prototyping is that the prototype may not get thrown away. Guimaraes observes this phenomenon "creating trouble at some companies" when "undocumented prototypes that were intended to be thrown away are kept, and become the poorly planned bases for large, complex systems that are consequently difficult to use and maintain" [Gui87]. Managers can avoid this pitfall by adequately training the programmers and maintaining a firm commitment to the prototyping paradigm. Careful definition of the scope and purpose of the prototype is also indicated as a means of avoiding this pitfall.

Lengthy iteration of the prototype phase

Prototype development can be time consuming, especially when the purpose and scope of the prototype is not initially well-defined. Boar's work [Boa85] describes how inadequate narrowing of the scope of the prototype can lead to thrashing or aimless wandering, the result of which will be of little use to a design team later on. Four of the industry sources support Boar's claim [Ala84, CB85, HLC82, Tam82]. Additional suggestions for avoiding this pitfall include using a highly disciplined approach to scheduling prototyping activities such as described in [Hek87], and avoiding throwing entry-level programmers into a rapid prototyping environment [CB85].

Skeptical end-users

End-users can become skeptical when given too much access to the prototype. Users may equate the incompleteness and imperfections, which naturally exist in a prototype, with shoddy design.

By limiting user interaction to a more controlled setting, user expectations can be kept at reasonable levels [CB85]. Two sources [Ala84, A3] recommend not overselling the prototype.

Evolving a large system results in a large mess

Evolutionary prototyping on large projects can result in a system filled with patches as hastily-designed modules become the root of later problems. One way to avoid this is to use an object oriented approach [AB90, GCG+89, A5]. Another method is to limit prototyping to user interface modules which are less likely to involve important data structure design decisions. A highly disciplined approach such as that used in [Hek87] is also recommended.

Underestimating conversion time

Prototyping languages are often utilized to ease implementation of a particular aspect of the system. For example, if the prototype is developed to test various user interface options, a language which provides convenient I/O capabilities is selected. The conversion may be non-trivial if the ultimate target language does not have such simple I/O handling. This pitfall was observed by Zelkowitz [Zel80], and alluded to by Taenzer [Tae91]. Careful definition of the scope of the prototype, and a systematic comparison of the features of both languages can help to avoid this pitfall.

6 Conclusions

The real-world case studies suggest that rapid prototyping, when employed properly, leads to improved software quality. The primary improvements are ease of use, better match with user needs, tighter code, and often better maintainability. We do not find support for the common notion that rapid prototyping cannot be used for developing large systems. We find no particular bias towards either keep-it or throw-away prototyping, and the case studies provide little insight into which languages are better for prototyping. We find a number of inherent pitfalls with prototyping. In order to avoid these pitfalls (described in Section 5), we recommend that developers try the following:

- Carefully define the purpose and scope of the prototype.
- Avoid the use of entry-level programmers.
- Utilize a design checklist.
- Use an open system environment (such as object-oriented methods).
- Consider performance issues early.
- Limit end-user interaction to a controlled setting.
- Do not under-estimate conversion time.
- *Do not* keep a prototype that was not initially intended to be kept.

Case study data is not easy to find and is somewhat biased. Negative results are seldom published. Our analysis can be improved with additional case study data, especially descriptions of rapid prototyping in large software projects. Although we cannot conclude that rapid prototyping

is dangerous for large projects (the data indicates otherwise), we are not willing to state that rapid prototyping is good for large projects using information from only three case studies.

Rapid prototyping is being successfully employed in the software industry. With the lessons provided by the case studies, rapid prototyping can be used to improve software quality.

References

[Boa85] B. Boar. *Application Prototyping - A Project Management Perspective.* AMA Membership Publications Division, 1985.

[Bro75] F. Brooks. *The Mythical Man-Month.* Addison-Wesley, 1975.

[Har87] K. Harwood. On prototyping and the role of the software engineer. *ACM SIGSOFT Software Engineering Notes*, 12(4):34, Oct 1987.

[LB89] J. Leszczykowski and J. Bieman. Prosper: A language for specification by prototyping. *Computer Languages*, 14(3):165–180, 1989.

[Mit82] R. Mittermeir. Hibol – a language for fast prototyping in data processing environments. *ACM SIGSOFT Software Engineering Notes*, 7(5):133–141, Dec 1982.

[Pat83] B. Patton. Prototyping – a nomenclature problem. *ACM SIGSOFT Software Engineering Notes*, 8(2):14–16, April 1983.

[Rat88] B. Ratcliff. Early and not-so-early prototyping – rationale and tool support. *Proc. COMPSAC 88*, pp 127–134, Nov 1988.

Case Studies

[AB90] E. Arnold and D. Brown. Object oriented software technologies applied to switching system architectures and software development processes. *Proc. 13th Annual Switching Symposium*, vol 2, pp 97–106, 1990.

Describes a methodology for applying object oriented technology to switching systems. The advantages of an object oriented approach as it relates to prototyping is discussed.

[Ala84] M. Alavi. An assessment of the prototyping approach to information systems development. *Communications of the ACM*, 27(6):556–563, June 1984.

Twelve information systems development projects using the prototyping approach in six organizations are analyzed. Also, an experiment involving nine student groups is presented which compares the use of the prototyping method with the specification approach.

[BGS84] B. Boehm, T. Gray, and T. Seewaldt. Prototyping versus specifying: A multiproject experiment. *IEEE Transactions on Software Engineering*, SE-10(3):290–302, May 1984.

Seven student teams developed the same product, three use prototyping, and four use specification. Boehm compares the specification technique with the prototyping technique.

[BJK+89] P. Bonasso, P. Jordan, K. Keller, R. Nugent R. Tucker, and D. Vogel. A software storming approach to rapid prototyping. *Proc. 22nd Annual Hawaii Conf on System Sciences*, vol 2, pp 368–376, 1989.

A new methodology for rapid prototyping called *software storming* is proposed, which involves an intense one-week period of videotaped interaction between software designers and end users.

[CB85] J. Connell and L. Brice. The impact of implementing a rapid prototype on system maintenance. *AFIPS Conference Proceedings*, vol 54, pp 515–524, 1985.

Describes case studies using the INGRES data base system. One project was completed and implemented to the apparent satisfaction of all parties. Another was criticized by various parties and eventually abandoned.

[FM82] R. Ford and C. Marlin. Implementation prototypes in the development of programming language features. *ACM SIGSOFT Software Engineering Notes*, 7(5):61–66, Dec 1982.

Describes the authors experiences applying prototyping techniques to the development of programming languages with advanced features.

[GCG+89] R. Gupta, W. Cheng, R. Gupta, I. Hardonag, and M. Breuer. An object-oriented VLSI CAD framework. A case study in rapid prototyping. *IEEE Computer*, 22(5):28–36, May 1989.

Shows the suitability of rapid prototyping for the development of CAD systems, and to object oriented development. The focus is on the specific application being prototyped, rather than the merits of prototyping in general.

[GHPS79] G. Groner, M. Hopwood, N. Palley, and W. Sibley. Requirements analysis in clinical research information processing – a case study. *IEEE Computer*, 12(9):100–108, Sept 1979.

A number of small clinical research programs were developed and evaluated for users who have difficulty in clearly expressing their computing needs.

[Gom83] H. Gomaa. The impact of rapid prototyping on specifying user requirements. *ACM Software Engineering Notes*, 8(2):17–28, April 1983.

Describes a case study of a large system developed at General Electric called PROMIS, a Process Management and Information System for integrated circuit fabrication. The progress of prototyping and subsequent development are tracked by the author.

[GS81] H. Gomaa and D. Scott. Prototyping as a tool in the specification of user requirements. *5th Int. Conf on Software Engineering*, pp 333–339, 1981. See [Gom83] for description.

[Gui87] T. Guimaraes. Prototyping: Orchestrating for success. *Datamation*, pp 101–106, Dec 1987.

Describes an extensive field study of 48 Fortune 1000 companies which evaluates whether or not prototypes should be discarded. Although the majority of companies employ throw-away prototyping, the author concludes that generally keep-it prototyping is preferable.

[Hek87] S. Hekmatpour. Experience with evolutionary prototyping in a large software project. *ACM SIGSOFT Software Engineering Notes*, 12(1):38–41, Jan 1987.

Rapid prototyping methodologies are used to develop a prototyping language called EPROL and a prototyping system called EPROS. Shows the successful use of evolutionary prototyping on large systems.

[HLC82] C. Heitmeyer, C. Landwehr, and M. Cornwell. The use of quick prototypes in the secure military message systems project. *ACM SIGSOFT Software Engineering Notes*, 7(5):85–87, Dec 1982.

A small military prototype messaging system is developed using LISP. Focuses on the role of the prototype, prototyping efforts in general, and reuse of prototype code.

[Rze89] W. Rzepka. A requirements engineering testbed: Concept, status and first results. *Proc. 22nd Annual Hawaii Conference on System Sciences*, vol 2, pp 339–347, 1989.

Describes a prototyping environment at the Rome Air Development Center. Concentrates on a heavily I/O intensive application, and notes significant speedup in development time.

[SJ82] E. Strand and W. Jones. Prototyping and small software projects. *ACM SIGSOFT Software Engineering Notes*, 7(5):169–170, Dec 1982.

The retention of the prototype code and the use of a special-purpose prototyping language are shown to be useful techniques in small-scale software projects.

[Tae91] D. Taenzer (U.S. West). Personal communications.

Rapid prototyping improved quality and ease of use of final products, and increased user participation. Developers are often pressured into reusing a throw-away prototype. Careful definition of the scope and definition of the prototype is recommended.

[Tam82] D. Tamanaha. An integrated rapid prototyping methodology for command and control systems: Experience and insight. *ACM SIGSOFT Software Engineering Notes*, 7(5):387–396, Dec 1982.

Describes a major ($100M) military project implemented successfully using rapid prototyping. CICS is used as the development environment. Provides insights into the effects that prototyping necessitates on management techniques and the software development process.

[Zel80] M. Zelkowitz. A case study in rapid prototyping. *Software – Practice and Experience*, 10(12):1037–1042, Dec 1980.

Describes experiences implementing Backus' FFP System using rapid prototyping. The need for certain functionality became apparent while exercising the prototype.

Sources Requesting Anonymity

[A1] Employee at major university.

Development of a University online registration system is described. Authors report improved customer satisfaction, ease of use, and ease of training.

[A2] Researcher at major university.

Expressed satisfaction with SCHEME as a prototyping language, based on the ultimate production of a working system in C.

[A3] Engineer at large telecommunications firm.

Describes problems with rapid prototyping which initially stemmed from management not understanding the limits of a prototype, and which have caused hardships and ultimate failure.

[A4] Engineer at large military contracting firm.

A substantial improvement in product quality, reduced effort, lower maintenance costs, and faster delivery is achieved by the use of rapid prototyping. In particular, leveraging with off-the-shelf products helps greatly.

[A5] Engineer at large data processing firm.

Prototyping has been quite effective. Recommendations include using object oriented approach, throw-away prototyping, and the careful selection of an appropriate prototyping language.

[A6] Engineer at large Government/Military division.

Small government systems have been developed successfully with rapid prototyping. Reuse of 50% of the prototype was generally achieved. Product quality was improved, and users were more likely to get what they wanted. Some people became upset when their ideas were quickly discredited by experiences with the prototype.

[A7] Engineer at small software company.

Prototyping worked well in a small project environment. Lines-of-code per day can be bid at a higher rate, but the method only works if experienced engineers are available.