# Rapid Prototyping: Lessons Learned

V. Scott Gordon James M. Bieman

Department of Computer Science
Colorado State University
gordons@cs.colostate.edu, bieman@cs.colostate.edu

Preprint of article published with further revisions in *IEEE Software*, 12(1):85–95, January 1995. Reprinted in *Software Project Management: Readings and Cases, First Edition.* by C. Kemerer, Irwin Publishing, 1997. Translated into the Dutch and republishsed as Rapid prototyping: lessen uit de praktijk. *IT Management [Select]* 2(1): 42–59, June 1996.

**Abstract**

Rapid prototyping is a development method that may or may not be effective in improving software products and process. Assessing the effectiveness of rapid prototyping requires empirical data. We analyze 39 published and unpublished "real world" case studies of the use of rapid prototyping for developing software products. By identifying effects mentioned in multiple sources, we are able to extract information about software products and processes resulting from the use of prototyping, as well as potential difficulties. We find that, with careful planning and management, software developers can effectively use rapid prototyping.

## 1 Introduction

The selection of an appropriate lifecycle paradigm is crucial to the successful development of software systems. Although the "waterfall" model remains the most commonly emphasized paradigm, there is a continuing interest in evolutionary methods such as rapid prototyping. The notion of a prototyping approach to software development has been widely known for over 15 years, since shortly after the publication of Brooks' *The Mythical Man-Month* [Bro75]. It is now time to assess the effectiveness of the use of rapid prototyping. This report describes an ongoing study, now in its third year, which examines the use of rapid prototyping.

Rather than conduct a controlled study of our own, we accumulate and compare the results reported in as many rapid prototyping case studies as we can find. Although many books and research papers on rapid prototyping have been published (including [Boa85, BKM84, Bud92, CS89]), few report on actual real-world experience. We have found 23 published case studies that include information regarding the effectiveness of the technique. In order to broaden our perspective and increase the sample size, we solicited first-hand accounts. We used the internet news service, and sent questionnaires to individuals who claimed to have been involved in software development projects which utilized rapid prototyping. Many of the respondents required anonymity, but a few did not. In total, we found 39 sources of case study information to analyze.

The case studies report on the actual use of rapid prototyping in a variety of settings such as military, commercial, and system applications. We examine the case studies for common experiences and opinions, and then tally the commonalities to identify those experiences which appear as recurring themes. We have previously used this method to report some of the effects of rapid prototyping on software quality [GB91, GB92]. Here we offer more comprehensive findings to include not only effects on software products, but also effects on the software *process*, such as effort, costing, etc. We also offer guidelines to help software developers avoid problems which the authors of the case studies experienced, or which they recognized and were able to avoid.

## 2 Rapid Prototyping

*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. Prototyping can give both the engineer and the user a chance to "test drive" software to ensure that it is, in fact, what the user needs. Alternatively, engineers may utilize prototyping to improve their understanding of the technical demands upon, and the consequent feasibility of, a proposed system. The use of prototyping has been recommended as a way of correcting weaknesses of the traditional "waterfall" software development life cycle, by clarifying important system requirements to software developers and end users, *before* a full system is implemented.

Although case studies utilize various terminology, we try to adhere to the definitions suggested by Patton [Pat83] and Ratcliff [Rat88]. *Rapid prototyping* is prototyping activity which occurs early in the software development life cycle. Since we are only considering early prototyping, we use the terms "prototyping" and "rapid prototyping" interchangeably. There are two prototyping methodologies: throw-away and evolutionary. *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, we concentrate here solely on rapid prototyping.

## 3 Case Study Analysis

The case studies included in our analysis describe particular software projects implemented via rapid prototyping, and also discuss how the use of prototyping helped and/or hindered development. We identified the attributes (effects) described in three or more of the sources, and use these attributes in our analysis. In general, we use the attributes as defined by the case studies. The attributes were then tallied, along with relevant opinions, observations, and suggestions. We are particularly interested in finding observed differences between throw-away and evolutionary prototyping. Definitions do vary between sources, and this is a limitation of the study.

We had no control over the data collected or manner of reporting in the published case studies, which was often incomplete. We had better control over the completeness of the information in the firsthand accounts, since the respondents were asked to answer certain questions. Case studies vary in degree of rigor. Four of the sources observe multiple projects and present conclusions based on careful quantitative measurements of the results. More often, the cases offer subjective conclusions and suggestions in a less specific, more qualitative manner, acquired from personal experience in one project. Some of the studies include a minimal amount of quantitative measurement interspersed with subjective judgement. The case studies have varied objectives and intended audiences. Still, by comparing the results reported by the different studies, we found that we could extract important information. For example, one study may report difficulty with a particular rapid prototyping activity, while another study may suggest a remedy for the same problem. *We emphasize conclusions that were reached by multiple sources independently.*

We divide the common attributes into three categories: *product* attributes, *process* attributes, and *problems*. For example, "ease of use" is a product attribute. When an effect on one of the attributes is noted in a case study, we tally it along with any relevant explanation offered by the author. Some of our terminology must necessarily remain general because the authors concentrate on different details, and because the software systems described in the case studies themselves are so diverse. "Design quality," for example, can mean many different things depending on the nature of the system, or the point of view of the designer. 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 the intersection of very specific attributes between case studies is small, many of the case studies discuss certain general attributes (i.e., design quality, performance, etc.). We report those attributes which are included in many of the case studies. Although we could include additional attributes, or subdivide the attributes into more specific sub-attributes, the results would be less clear because there would be fewer common instances in the case studies. We do not include attributes that are not discussed in the case studies, even those that we think might be useful. The existence of other attributes not discussed in this study indicates that there was insufficient data on those attributes, and is an unfortunate limitation of the case study data.

The product attributes most commonly listed in the case studies are: *ease of use, match with user needs, performance, design quality, maintainability,* and *number of features*. Although design quality and maintainability are closely related, many of the cases reported design quality and maintainability separately, either because they had an opportunity to observe maintenance on the system, or because of other considerations such as the existence of maintenance tools.

The process attributes most commonly listed in the case studies are: *effort, degree of end-user participation, cost estimation*, and *expertise requirements,*. Commonality in listing process attributes is less than for product attributes, perhaps because prototyping itself is a process, and therefore it may not occur to authors to report other process effects.

The problems most commonly discussed in the case studies are those associated with: *large systems, maintenance, performance, delivering a throw-away prototype, end-user misunderstandings, budgeting, prototype completion,* and *conversion time*. While it is rare that case study authors actually describe occurrences of the problems, many authors describe steps that they took to avoid them. When a source does describe one of the problems, often they did not employ one of the suggestions mentioned in another source. We collect the relevant suggestions and include them with the descriptions of the individual issues. We provide information on the conditions under which a particular problem may or may not need to be purposefully avoided. Our overall objective is to use the case studies to develop guidelines for effective use of rapid prototyping.

## 4 Sources of Case Study Data

Industry use of rapid prototyping appears to be a recent phenomenon. We located 23 published reports containing 25 case studies. The earliest is from 1979, while most are from the mid-to-late 1980's. We also found three papers which analyze other rapid prototyping cases. To supplement the published reports, we collected questionnaires and personal accounts from fourteen individuals. Thus, we have a total of 36 sources describing 39 cases. The case study sources are listed in Figure 1. The sources represent a variety of organizations: AT&T, GE, RAND, MITRE, Martin Marietta, Los Alamos, Tektronix, ROME, Hughes, government divisions, and others. The fourteen personal accounts are listed as "anonymous sources" due to requests for anonymity. Some cases involve separate prototyping and development teams. Figure 2(a) shows, in graphical form, the distribution of case study sources. Ten of the sources are projects conducted at Universities, but only three of these are student projects. Twelve of the sources describe military projects. The remaining 17 describe other professional software development.

## Figure 1: Published Case Studies

1. Alavi. An assessment of the prototyping approach to information systems development. *Comm ACM*, June 1984.
2. Arnold and Brown. Object oriented software technologies applied to switching system architectures and software development processes. *Proc. 13th Annual Switching Symposium*, 1990.
3. Barry. Prototyping a Real-Time Embedded System in Smalltalk. *OOPSLA '89*.
4. Boehm *et.al.*, Prototyping versus specifying: A multiproject experiment. *IEEE Trans. on Software Engr*, May 1984.
5. Connell and Brice. The impact of implementing a rapid prototype on system maintenance. *AFIPS*, 1985. (2 cases)
6. Ford and Marlin. Implementation prototypes in the development of prog. lang. features. *ACM SE Notes*, Dec 1982.
7. Gomaa. The impact of rapid prototyping on specifying user requirements. *ACM SE Notes*, April 1983.
8. Goyden. The Software Lifecycle with Ada: A Command and Control Application. *Tri-Ada '89*, 1989.
9. Groner *et.al.*, Requirements analysis in clinical research info. processing – a case study. *IEEE Computer*, Sept 1979.
10. Guimaraes. Prototyping: Orchestrating for success. *Datamation*, Dec 1987.
11. Gupta *et.al.*, An obj-oriented VLSI CAD framework – A case study in rapid prototyping. *IEEE Computer*, May 1989.
12. Heitmeyer *et.al.*, The use of quick prototypes in the secure military msg. systems project. *ACM SE Notes*, Dec 1982.
13. Hekmatpour. Experience with evolutionary prototyping in a large software project. *ACM SE Notes*, Jan 1987.
14. Jordan *et.al.*, Software Storming - Combining rapid prototyping and knowledge eng. *IEEE Computer*, May 1989.
15. Junk *et.al.*, Comparing the Effectiveness of Software Dev. Paradigms: Spiral-Prototyping vs. Spec. *PNSQ*, 1991.
16. Kieback *et.al.*, Prototyping in Industrial Software Projects. *GMD-Studie* Nr. 184 (in German, a translated version will appear in *Information Technology and People*). (describes 5 cases, 3 of which are used in this study)
17. Luqi. Software Evolution through Rapid Prototyping. IEEE Computer, May 1989.
18. Martin *et.al.*, Team-Based Incremental Acquisition of Large-Scale Unprecedented Sys. To appear in *Policy Sciences*.
19. Rzepka. A requirements eng. testbed: Concept, status and first results. *22nd Hawaii Conf. on Sys. Sci.*, 1989.
20. Strand and Jones. Prototyping and small software projects. *ACM SE Notes*, Dec 1982.
21. Tamanaha. An integrated rapid prototyping methodology for command and control sys. *ACM SE Notes*, Dec 1982.
22. Zelkowitz. A case study in rapid prototyping. *Software – Practice and Experience*, Dec 1980.

## Anonymous Sources

a. *Employee at major university*. Development of a University online registration system.

b. *Researcher at major university*. Development of a campus support system using the SCHEME prototyping language.

c. *Engineer at large telecommunications firm*. Language development.

d. *Engineer at large military contracting firm*. Medium-large system.

e. *Engineer at large data processing firm*. Industrial application.

f. *Engineer at large Government/Military division*. Development of small aerospace systems.

g. *Engineer at small software company*. Development of low-level system software.

h. *Engineer at small Government/Military contractor*. Uses special-purpose prototyping tools.

i. *Engineer at large manufacturer*. Workstation development.

j. *Engineer at large communications and control firm*. Contract software development.

k. *Consultant for a defense contractor*. Internal support software.

l. *Engineer at large electronics firm*. Operating system development using the RAPID prototyping language.

m. *Engineer at small software company*. Internal support software.

n. *Engineer at large communications firm*.

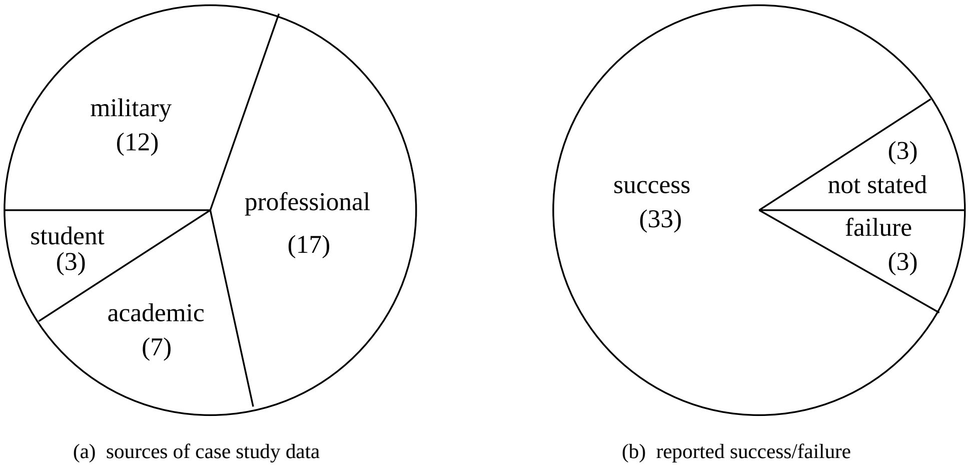


Figure 2: Information concerning case studies

Rapid prototyping is deemed a success in 33 of the 39 cases (see Figure 2(b)). Of the remaining six, three were described as failures and three did not claim success or failure. Although 85% of the case studies report success, we expect that there is some bias in the data since failures are seldom reported. Some of the (successful) sources, however, address intermediate difficulties encountered and perceived disadvantages of rapid prototyping. Six of the sources describe projects which involve no customer; the goal of these projects is the development of a system to be used by the developers. We avoid drawing strong conclusions regarding clarity of requirements or successful analysis of user needs when a project does not involve a separate user.

## 5 Effects on Software Product Attributes

Figure 3 shows six commonly-mentioned areas in which prototyping affected product attributes observed in the final system. For each effect, the figure indicates which case studies observe either a positive or negative impact. The case studies are referenced either by a number (for published studies) or a letter (for anonymous sources). The case studies are listed by number/letter in Figure 1. We also indicate the relative number of studies in which the particular effect was not observed or was not discussed. For each effect, two comments from the case studies are included to illustrate some of the firsthand experiences. The relatively high number of unreported effects reflects the diversity of reporting methods among the case studies.

### 5.1 Usability factors

Users have an opportunity to interact with the prototype, and give direct feedback to designers. Sometimes users are not sure that they want certain functions implemented until they actually can try them. Further, *the need for certain features* may not be apparent until actual use exposes an omission or inconvenience. Users may also find certain features or terminology confusing. Thus it is logical that prototyping tends to help ensure that the first implementation (after the prototype) will meet users needs, especially when the prototype includes the user interface. These findings are consistent with Brooks' famous maxim, "plan to throw one away; you will, anyhow" [Bro75]. That is, the first attempt at developing a system will likely fail to meet user needs, and be discarded. It is better that the first effort be a prototype rather than a final deliverable.

Figure 3: Software Product Effects (with selected comments) reported in 39 case studies.

| | improved (17) | not stated |
|---|---|---|
| **Ease of Use** | 1,4,7,11,12,15,16,20,22,a,c,d,f,i,j,m,n | |

‘‘[the author] soon tired of retyping in definitions for each ... run’’ [22]

‘‘Misunderstandings between the software developers and users were revealed.’’ [7]

| | better matched (22) | worse | not stated |
|---|---|---|---|
| **User Needs** | 1,5*,7,8,9,11,14,15,16*,18,20,21,22,d,f,h,i,j,l,m,n | 5* | |

‘‘Omissions of function are ... difficult ... to recognize in formal specifications.’’ [20]

‘‘Prototyping helps ensure that the nucleus of a system is right ... ’’ [1]

| | better | worse | not stated |
|---|---|---|---|
| **Performance** | 16,17 | 5,11,15,i,j,l | |

‘‘ ... consider performance as early as possible if [it] is to evolve into the final system.’’ [11]

‘‘We have had some performance problems that may be related to ‘design baggage’ ... ’’ [i]

| | better | worse | not stated |
|---|---|---|---|
| **Design Quality** | 3,5*,6,8,13,15,16,h | 4,5*,21,j,l | |

‘‘... allows early assessment of ... techniques required to implement specific features.’’ [6]

‘‘Missing from the documentation was a ... design of system procedures and control flows.’’ [5]

| | easier | harder | not stated |
|---|---|---|---|
| **Maintainability** | 3,4,5*,13,18,d,h,n | 5*,11,15,i,j,l | |

‘‘[if] no work was done ... for a prolonged time, maintenance may be a real problem.’’ [i]

‘‘... can reduce the ‘maintenance’ associated with changing requirements.’’ [n]

| | increase | decrease | not stated |
|---|---|---|---|
| **# of Features** | 15,h,j,l | 4,5* | |

‘‘... fostered a higher threshold for incorporating marginally useful features.’’ [4]

‘‘The customer ... goes crazy adding features and making changes.’’ [h]

* -- *these sources describe multiple case studies.*

The effect of prototyping on the number of features in a final system is less clear. The intuitive notion that the prototyping paradigm gives the end user a license to demand more and more functionality is not entirely borne out by the case studies. Several sources report that prototyping caused critical components to be stressed, and non-critical features to be suppressed, thus *reducing* the total number of features. It was slightly more common, however, that the number of features increased. This was observed for three different reasons: (1) special-purpose prototyping languages make it easy to add new features, (2) internal software development sometimes requires less time to determine baseline requirements, allowing more time to consider additional features, and (3) users demanding more and more functionality. Contractors could consider using cost add-ons for additional functionality as a counter-incentive if excessive user demands are anticipated.

### 5.2 Structural factors

Structural factors include those related to design quality, maintainability, and performance. Here the effect of prototyping has a greater chance of being negative. We shall discuss each in turn.

The effect of rapid prototyping on system performance depends partly on the scope of the prototype. When the prototype focuses solely on the user interface, system performance is likely to be unaffected (although there are a few cases, described in Section 7, where performance can be affected). When the purpose of the prototype is to examine various design alternatives, performance can be affected in a variety of ways. Sometimes prototyping can lead to better system performance, since it is easier to test several design approaches. However, evolutionary prototyping can lead to problems when performance is not adequately measured and either: (1) inefficient code is retained in the final product, or (2) the prototype demonstrates functionality that is unrealizable under normal usage loads. The case studies contain more evidence of performance problems for evolutionary prototypes than for throw-aways. Section 7 describes the possible problems and suggests ways to improve performance.

Design quality effects are also mixed. More sources indicate an improvement in design quality, both for evolutionary and throw-away prototyping. However, design problems are more commonly observed among evolutionary prototyping cases. Some sources report that evolutionary prototyping can result in a system with a less coherent design and more difficult integration. Other sources state that the multi-stage design/modify/review process can result in a better overall design. Several sources indicated that the code produced was longer (although a few state the reverse), but these same sources also noted that this was not necessarily good or bad.

Quality also suffers when, during evolutionary prototyping, design standards are not enforced in the prototype system. Even when using good tools, design can suffer when remnants of discarded design alternatives are not physically removed. To avoid these problems, it is useful to employ a design checklist that each section of incorporated code must satisfy. 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, subject to the limitations discussed previously. A problem associated with throw-away prototyping is that a prototype which was intended to be thrown-away might actually be kept. This is discussed in greater detail in Section 7.

Maintainability effects are similar to those observed for design quality, but maintainability problems can be quite troublesome. Again, more sources cite improvement in overall maintainability. But for evolutionary prototyping, slightly more sources observe reduction in maintainability. Possible solutions to maintainability problems are described in Section 7.

Some of the reports describe successful maintenance of prototyped systems, even for very substantial size projects. The high degree of modularity required for *successful* evolutionary prototyping can generate easily maintainable code, because such a system is more likely to be built out of reusable and replaceable functional modules. There are also indirect reductions in maintenance costs owing to the greater likelihood that user needs will be met *the first time*. "Maintenance" often involves correcting invalid requirements, or responding to changing requirements. Rapid prototyping can help reduce this sort of maintenance since it is likely that user needs will have been more completely met.

### 5.3 Throw-Away vs. Evolutionary prototyping

Overall, we find that software product effects are generally positive, with certain problems related primarily to evolutionary prototyping. While many engineers are adamantly opposed to evolutionary prototyping (often citing Boar's books which generally recommend against evolutionary prototyping [Boa85]), examination of the case studies paints evolutionary prototyping in a more positive light, even for substantial software projects. Further, some sources suggest that, for small projects, throw-away prototyping (compared to evolutionary) is economically infeasible. However, we cannot ignore that quality attributes such as performance, design quality, and maintainability can suffer during evolutionary prototyping if steps are not taken to avoid the relevant problems (see Section 7). Figure 4 shows the relative number of case studies using throw-away and evolutionary prototyping, and Figure 5 gives a breakdown of the effects of paradigm choice on performance, design quality, and maintainability.

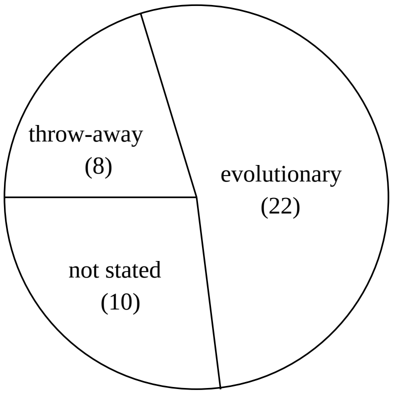


Figure 4: Distribution of case studies by paradigm

## 6 Effects on Software Process Attributes

Figure 6 lists four commonly mentioned areas in which prototyping affected process attributes. For each effect, the figure indicates which case studies observe either a positive or negative impact, and additional information as in Figure 3. Again, the relatively high number of unreported effects reflects the diversity of reporting methods among the case studies.

Although the case studies discuss many other process effects, there is less commonality than for product effects. Therefore only four attributes are included in Figure 6. We have included some discussion of other effects, especially those concerning language selection. But the lack of commonality makes inclusion inappropriate.

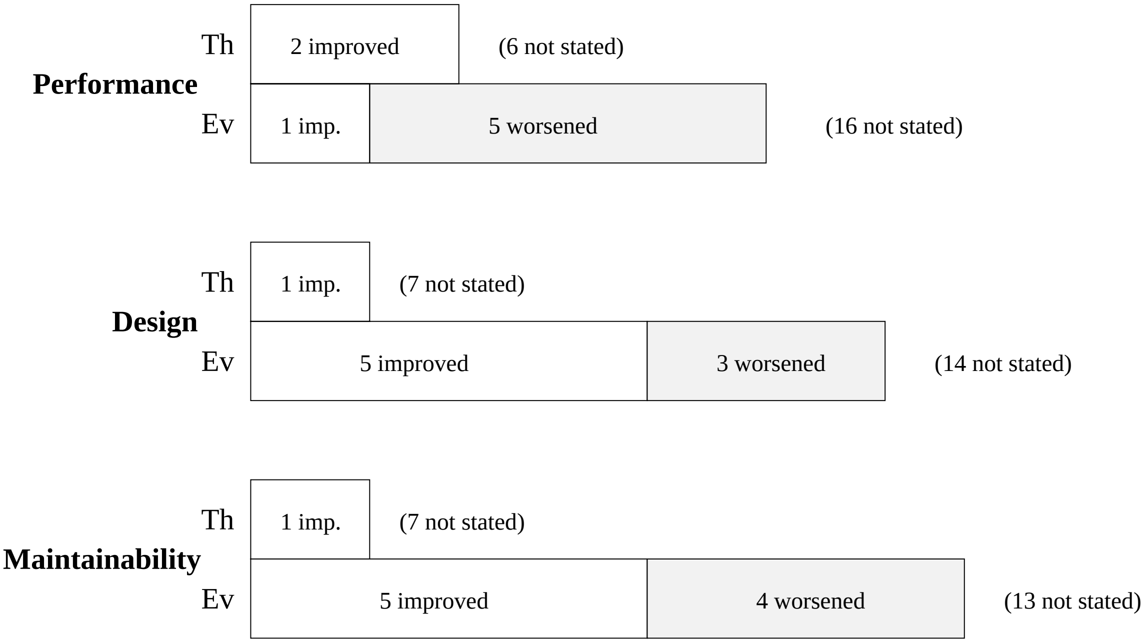


Figure 5: Throw-away vs. evolutionary effects on reported projects

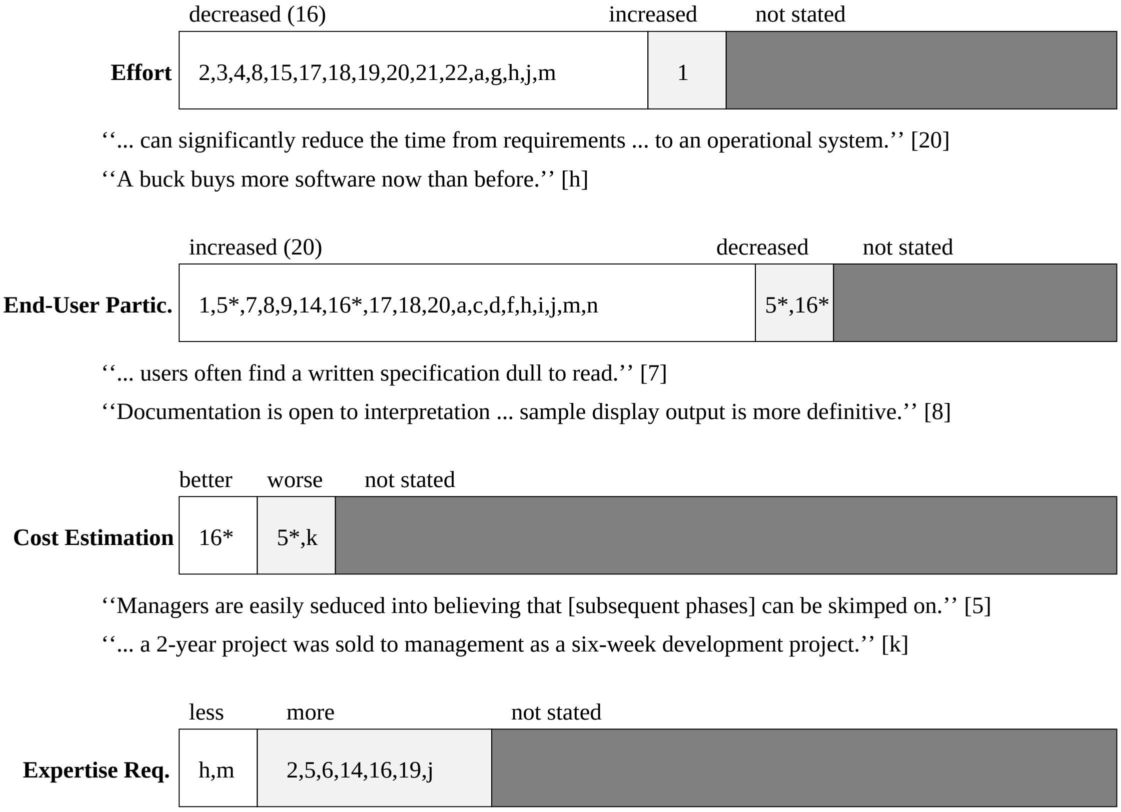
### 6.1 Effort and estimating effort

One of the most commonly cited benefits of rapid prototyping is that it can lead to a decrease in effort. Most of the case studies support this notion. In some cases, the decrease in effort is dramatic. One source reports effort reduction by a factor of 3.5, another a reduction of 45%. One military project observed a productivity of 34 lines of code per day per programmer, more than six times the estimate for typical military software systems. There are various reasons for the decrease in total effort. Faster design is possible when requirements are clearer or more streamlined. Also, in the case of evolutionary prototyping, portions (or all) of the prototype can be leveraged, causing overlap in the requirements effort and the development effort.

There are a few cases where development effort increased. The lack of an organized methodology has been suggested as a possible cause of wasted effort, although it is not clear whether this effect was actually observed. Sometimes a prototype can reveal that needs are greater than first thought, resulting in greater development effort. But it is unfair to call this a case of increased effort, because actually the use of prototyping here has *prevented significant wasted effort*.

There is greater skepticism surrounding contract bidding (i.e., estimating effort beforehand) in a rapid prototyping environment. Most of the case studies do not address this issue in much detail. In a few cases, the early availability of visible outputs can cause users and managers to be easily seduced into believing that the subsequent phases will be easy to complete. As a result, projects can be underbid. This underbidding could possibly be avoided with proper training of managers in prototyping methodology. Underbidding is discussed in greater detail in Section 7.

Some cases utilized rapid prototyping as a separately costed "proof-of-concept" item. In this scenario, a project may prove to be infeasible or not cost-effective. Such feasibility and cost information provides a mechanism for a customer to abandon a project at reasonable expense. Developers using proof-of-concept prototyping may have to bid actual development of a complete system separately. Prototyping does give the developer a chance to abandon a project that might have been under-bid in a specification environment. Boar [Boa85] gives a number of suggestions for keeping cost estimates



‘‘... can significantly reduce the time from requirements ... to an operational system.’’ [20]

‘‘A buck buys more software now than before.’’ [h]

‘‘... users often find a written specification dull to read.’’ [7]

‘‘Documentation is open to interpretation ... sample display output is more definitive.’’ [8]

‘‘Managers are easily seduced into believing that [subsequent phases] can be skimped on.’’ [5]

‘‘... a 2-year project was sold to management as a six-week development project.’’ [k]

‘‘Prerequisites to successful prototyping include ... knowledgeable users and designers.’’ [1]

‘‘The best way to clear up misconceptions ... is with an in-house training program.’’ [5]

* -- *these sources describe multiple case studies.*

Figure 6: Software Process Effects (with selected comments)

under control. However, we find insufficient case study data to draw conclusions concerning the use of prototyping as it relates to cost estimation.

## 6.2 Human factors and staffing considerations

Increase in user participation (in the requirements definition phase) is commonly observed among the case studies. Users are more comfortable reacting to a prototype than reading a “boring” written specification. Increased user participation, as described above, has a positive effect on the software product by increasing the likelihood that the user’s needs will be met. In fact, lack of sufficient user participation can negate some of the benefits of rapid prototyping. One source describes a case where the customer’s management purposely excluded end users from interacting with the prototype, so that inappropriate allocation of personnel (i.e., in a particular division’s favor) would not be revealed for as long as possible. Another source observed the same phenomenon, and referred to this as “staff rationalization”. This dangerous political maneuver can be avoided simply by making sure that end users remain actively involved, and thus become fully aware of the nature of the problem at hand. Since one of the main advantages of rapid prototyping is in revealing the actual requirements, developers should insist on prototype interaction by end users, not just middle management.

Several sources recommend an experienced, well-trained team as essential for successful prototyping, because prototyping often requires overlap of design decisions with programming tasks. Problems can result when inexperienced team members are put in the position of having to make high-level design decisions. Examination of the case studies lends support to this concern. One case specifically describes a project that failed in part because temporary student programmers were thrown into a rapid prototyping environment. Other case studies indicate that successful use of prototyping would not have been possible without highly experienced engineers. Two cases utilized entry-level programmers successfully in a rapid prototyping environment, and attributed this success to the availability of good prototyping tools. Overall, the evidence suggests that it may be dangerous to throw inexperienced programmers into a rapid prototyping environment, especially when the prototyping activities require high-level design decisions.

### 6.3 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 (e.g., Interface Builder, RAPID, etc.), and 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 affects the software process at many levels. These effects can be positive or negative depending on other factors, such as the nature of the system being developed.

Although most sources stress the importance of carefully selecting a language suitable for prototyping, 38 cases employed 26 different languages. The most popular single language choice was Lisp, although it was used in only four cases. Object-oriented methods are receiving increased attention for rapid prototyping uses [AGS89], and several of the cases attribute success to the use of object-oriented approach (three use Smalltalk). Six sources identify object-oriented methods as being particularly well-suited for prototyping and for avoiding certain problems.

## 7 Effective Use of Prototyping

Most of the case studies describe successful projects, so there is less direct data on prototyping problems. However, many authors explain their development approach in terms of anticipating and avoiding particular situations. In a few cases, a reported problem could have been avoided by employing one of the suggestions in another source. Thus we are able to locate common instances of problems and possible solutions described in several sources. We only describe problems related directly to the use of prototyping. That is, we concentrate on situations which are less likely to occur when using the specification approach.

### 7.1 Performance issues

It is often useful to prototype critical aspects other than the user interface. Designing the entire system starting from the user interface can be dangerous, since the user interface may not characterize the best overall system structure. Thus a user interface prototype should be considered a piece of a requirement specification and not a basis for system design. Again, discarding the prototype is also an option, but can only be done when the performance of the discarded prototype is not important (an invalid assumption if the purpose of the prototype is to evaluate the performance of a particular design). Thus, 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 way of avoiding this problem is to use an open system development environment to make it easier to integrate faster routines when necessary.

Some sources cite inferior performance due to the use of 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.

## 7.2 Avoiding 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 ultimate failure of a project. 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 period of time. Lack of knowledge of rapid prototyping techniques is not limited to engineers or managers. Sales staff may pass along inappropriate expectations to customers after seeing "working" prototypes. Users then understandably became skeptical or upset when told that development would take longer than they were led to believe. High user expectations were typically fueled by over-enthusiastic or under-controlled access to the prototype.

By limiting user interaction to a more controlled setting, user expectations can be kept at reasonable levels. Users should be clearly told that they are interacting with a mockup for purposes of requirements clarification, and not with a working system. In some cases, it might be desirable for interaction to be limited to specific sequences as administered by the developers. Further, developers should not "oversell" the prototype in an effort to impress the customer. Sales and managerial staff should be trained to properly understand (and convey) the nature and purpose of the prototyping phase and the prototype itself.

## 7.3 Improving code maintainability

Certainly, a prototype which is developed quickly, massaged into the final product, and then hurriedly documented can be very difficult to maintain or enhance. Further, failing to re-evaluate a prototype design before starting to implement the final system can result in a product which inherits patches acquired during the prototype phase. Documentation criteria should be included in the design checklist to ensure complete system documentation of the prototype. Other suggestions include frequent reviews and the use of object-oriented technology. Discarding the prototype is also an option if the thrown-away prototype code will not be needed.

Rapid prototyping can also have a negative effect on 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 the two languages. An increase in complexity can result, even when system design is good. A prototyped system can become impossible to maintain if it was developed using prototyping tools that are not available to the maintenance engineers.

## 7.4 Avoid delivering a "throw-away"

Very poor design quality can result when a prototype is meant to be thrown away, but is kept instead in order to save costs. This is a surprisingly common problem which 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. One of the perils of *throw-away* prototyping is that the prototype *may not get thrown away*. Managers can avoid this problem by maintaining a firm commitment to the prototyping paradigm, and by careful definition of the scope and purpose of the prototype.

## 7.5 Budgeting and the prototype

Three cases describe scenarios in which projects were underbid. Because visible outputs are quickly available, managers and salespersons may be easily seduced into believing that the subsequent phases can be skimped on. Overconfidence can result in underbidding, and prototyping can provide an environment which promotes overconfidence. In one case, a project originally estimated as requiring two years was modified to six weeks on the basis that prototyping would achieve fast, working results. Sales and managerial staff should be trained to properly understand the nature and purpose of the prototyping phase and the prototype itself, and the distinction between a prototype and a complete system.

Many questions surrounding bidding in a prototyping environment remain unanswered. Many companies bid the prototyping phase separately, sometimes as a "proof-of-concept" item. This may not be an appropriate solution for many situations. More data is needed on costing issues.

## 7.6 Completion and conversion of the prototype

Prototype development can be time consuming, especially when the purpose and scope of the prototype is not initially well-defined. Boar [Boa85] describes how inadequate narrowing of the scope of the prototype can lead to thrashing or aimless wandering. Six of the sources give evidence in support of Boar's claim. Suggestions for avoiding this problem include using a disciplined approach to scheduling prototyping activities, careful definition of the scope of the prototype, and avoiding throwing entry-level programmers into a rapid prototyping environment.

Prototyping languages are often utilized to ease implementation of a particular aspect of the system. For example, if the prototype is developed to test user interface options, a language which provides convenient I/O is selected. However, converting the prototype into the final system may require significant effort and time, and this problem is exacerbated when a separate prototyping language is used. Conversion may be non-trivial if the ultimate target language does not have such simple I/O handling. Another example is when an object-oriented language such as Smalltalk is used, and the target language does not have inheritance. Cost or time overruns were observed in a few cases. Careful definition of the scope of the prototype, and a systematic comparison of the features of both languages can help to avoid this problem. Of course, this problem does not occur if the same language is used for both the prototype and the final system.

## 7.7 Problems associated with large systems

There are widely differing opinions on what constitutes a "large" software system. In one case study, a 200-line system is described as large, whereas other authors might consider any program of that size to be very small. For this reason, we try to avoid defining "large", and instead refer to systems that are at least 100,000 lines of code to be *substantial*. Although some researchers might claim that 100,000 line systems are not large but medium-sized, few would argue that programs of that size are small. To distinguish between medium and small projects, we used whatever description was used in the case study reports (that is, we did not select an exact boundary). Figure 7 shows a breakdown of the projects by size.

We find no support for the common notion that *evolutionary* prototyping is specifically dangerous for large projects. In fact, every case involving substantial projects used evolutionary prototyping. However, the problems involved with evolutionary prototyping grow somewhat in proportion to the size of the system being prototyped.

Evolutionary prototyping on large projects can result in a system filled with patches as hastily-designed modules become the root of later problems. The problems described earlier for performance and maintenance issues perhaps become more pronounced as system size grows. The methods of

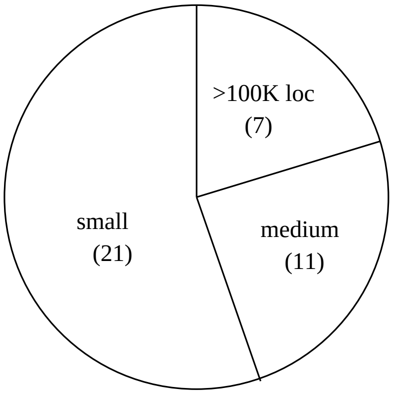


Figure 7: Distribution of case studies by project size

avoiding those particular problems are equally applicable here, such as using an object oriented approach, or limiting prototyping to user interface modules which are less likely to involve important data structure design decisions.

## 8 Conclusions

In a study of real world case studies of the use of rapid prototyping, we identify common observations and analyze the effects of the prototyping life cycle both on software products and on the software process. Our study was based on the reports of 39 published and unpublished relatively recent case studies, and represented a wide range of software development organizations including military/government, commercial, and academic developers.

Most of the case studies report that prototyping was successful; only three studies report failures (possibly because reports of failures are seldom made public). The product improvements most clearly identified by the case studies are an improved match with users needs and improved “ease of use” (see Figure 3). The most commonly cited positive effect on software products, mentioned by 22 of the case studies, is that prototyping was helpful in improving the capability of the product to satisfy the needs of users; only one study reports a negative effect. Prototyping improved product ‘ease of use’ in 17 of the studies; no studies report negative effects. The case studies report mixed results concerning the effects of prototyping on design quality and product maintainability. The only product attribute that, when mentioned, generally shows a negative effect is performance. Six cases note negative performance effects, while only two cases note performance improvements. Overall, the noted effects on product quality are positive.

No support is found for the common notion that rapid prototyping is not appropriate for large systems. We find no bias towards either keep-it or throw-away prototyping. Because of the variety of languages used in the case studies, we cannot draw any conclusions concerning the relative merits of prototyping languages.

The case studies report generally positive effects of prototyping on the software process (see Figure 6). Sixteen of the studies note decreased development effort; only one study reports an increase in effort. Twenty case studies report increased participation of end users; two cases report a decrease in such participation. However, seven of the cases report that greater expertise is required of developers, while only two cases report that prototyping requires less expertise. Reported effects on cost estimation are inconclusive.

A number of potential problems can result from the use of rapid prototyping. The most serious problems are poor design quality and maintainability (especially when using evolutionary prototyping), underbidding, and misunderstandings between developers and users. Potential problems can be avoided by carefully defining the purpose and scope of the prototype, and by not violating this defined purpose and scope. For example, a throw-away prototype should not be kept. Design and maintainability problems can be prevented through the use of design checklists, and by avoiding the use of entry-level programmers for making design decisions. Underbidding and misunderstandings can be prevented by limiting end-user interaction to a controlled setting, training sales and managerial staff to not oversell the prototype, and by not underestimating the time required to develop a product from a prototype.

We find that rapid prototyping can be successfully employed by the software industry in a variety of situations. The case studies dispel some often-held beliefs concerning prototyping. Rapid prototyping seems to have a number of realized benefits, and, when used properly, can improve the software development process and products.

# References

[AGS89] K. Auer, T. Goldstein, S. Sridhar, T. Love, and D. Thomas. Panel: From Prototype to Product!? *OOPSLA '89 Proceedings*, 482–484, Oct 1989.

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

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

[BKM84] R. Budde, K. Kautz, L. Mathiassen, and L. Züllighoven. *Approaches to Prototyping.* Springer-Verlag, 1984.

[Bud92] R. Budde. *Prototyping: an Approach to Evolutionary System Development.* Springer-Verlag, 1992.

[CS89] J. Connel and L. Shafer. *Structured Rapid Prototyping: an Evolutionary Approach to Software Development*. Yourdon Press, 1989.

[GB91] V. Gordon and J. Bieman. Rapid Prototyping and Software Quality - Lessons from Industry. *Pacific Northwest Software Quality Conference*, 1991, pp 19-29.

[GB92] V. Gordon and J. Bieman. Reported Effects of Rapid Prototyping on Industrial Software Quality. *Software Quality Journal*, 2(2):93-110, June 1993.

[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.